Clube Delphi +PHP

Ano 8 · Edição 101 · R$11,90

DevMedia group

Multi Camadas

Ótimas práticas para migrar sua solução client/server

Compre esta edição e ganhe:

Confira no portal ClubeDelphi PLUS, 2 vídeo-aulas:

- Como registrar em que folha foi impresso determinado registro
- ASP.NET 2.0 - Stored Procedures

PHP

Trabalhando com o MySQL e Firebird

Novidades Web

ASP.NET 2.0
Themes, Skins e CSS

WebParts
Construa portais customizáveis com o Delphi

Reaproveitamento de Código
Boas Práticas para garantir a reusabilidade de código

Ask the Expert
Do Excel para o StringGrid

Boas Práticas
Testes unitários com o Rad Studio

Notificando Erros ao Usuário
Crie um mecanismo elegante para indicar erros em formulários

ISSN 1517990-7

©2008 Microsoft Corporation. Todos os direitos reservados. Microsoft, Visual Studio, o logo do Visual Studio e "Seu potencial. Nossa inspiração" são marcas comerciais, registradas ou não, da Microsoft Corporation nos Estados Unidos e/ou em outros países. Os nomes das companhias ou produtos reais aqui mencionados podem ser marcas comerciais de seus respectivos proprietários.

Sumário

Olá, eu sou o **DevMan**! Desta página em diante, eu estarei lhe ajudando a compreender com ainda mais facilidade o conteúdo desta edição. Será um prazer contar com sua companhia! Confira abaixo o que teremos nesta revista:

Delphi — Web Novidades

12 – WebParts
Portais e páginas customizáveis com ASP.NET 2.0
[*Guinther Pauli e Adriano Santos*]

Delphi — Boas Práticas Projeto/Análise

18 – Testes unitários de software
Aprenda a efetuar testes unitários no Delphi 2007 com nUnit e csUnit
[*Marcelo Daibert e Marco Antonio Pereira Araujo*]

Delphi — Boas Práticas Expert

26 – Multicamadas
Ótimas razões para migrar sua solução client/server
[*Guinther Pauli*]

Delphi — Boas Práticas Boa Idéia

42 – Notificando erros de validação
Aprenda a aplicar técnicas de validação em sua aplicação
[*Paulo Quicoli*]

Delphi — Boas Práticas

48 – Reaproveitamento de código
Organizando o código comum da aplicação
[*Luciano Pimenta*]

Easy Delphi — Easy Delphi

54 – Themes, Skins e CSS no ASP.NET 2.0
Aprenda a trabalhar com temas e skins e mude a aparência de seus sites
[*Guinther Pauli e Adriano Santos*]

PHP — PHP

60 – Manipulação direta de dados
Aprenda comandos para manipulação de dados em SGBD com o PHP
[*Fabrício Desbessel*]

Você percebeu os nomes ao lado de cada matéria? Eles indicam o que você vai encontrar no artigo – dessa forma, você também pode ter uma idéia geral do que vai encontrar nesta edição como um todo! Os editores trabalham sempre no sentido de fechar a revista seguindo esta definição, para oferecer a você o melhor conteúdo didático!

Confira abaixo a lista com a definição dos tipos de artigo encontrados nesta edição:

[Mão na Massa] Artigos que focam na resolução de problemas - e não na tecnologia. A tecnologia empregada é apenas conseqüência. O artigo parte do pressuposto que o leitor já conhece a tecnologia, mas tem dificuldade de implementá-la em uma aplicação cotidiana. Por exemplo, um artigo que ao invés de "debulhar" todos os métodos e propriedades de um "Recordset", mostre como usar o Recordset de forma inteligente. para criar um carrinho de compras.

[PHP] Artigo com foco no PHP puro.

[Expert] Artigo com foco no leitor avançado.

[Boas Práticas] A Um dos objetivos da revista é levar para o leitor não somente as melhores técnicas, mas também as melhores práticas – Esse tipo de artigo foca em técnicas que poderão aumentar a qualidade do desenvolvimento de software.

[Web] Artigos sobre ou que envolvam técnicas de desenvolvimento para WEB.

[Novidades] Artigos sobre tecnologias de ponta, que ainda não fazem parte do dia a dia do desenvolvedor mas que prometem ser a próxima febre.

[Boa Idéia] A Esse tipo de artigo é quase um artigo "Mão na Massa" - a diferença é que o exemplo do artigo é tão criativo que é considerado uma 'boa idéia', dentro do contexto de desenvolvimento de software.

[Easy Delphi] Com foco no desenvolvedor iniciante.

Clube Delphi + PHP

Ano 8 - 101ª Edição - 2008 - ISSN 1517990-7 **Impresso no Brasil**

Corpo Editorial

Editor Geral

Guinther Pauli

guinther@devmedia.com.br

Editor Técnico

Adriano Santos

adrianosantos@devmedia.com.br

Equipe Editorial

Fabricio Desbessel, Maikel Scheid, Paulo Quicoli, Luciano Pimenta

Editor de Arte

Vinicius O. Andrade

viniciusoandrade@gmail.com

Revisão

Gregory Monteiro

gregory@clubedelphi.net

Distribuição

Fernando Chinaglia Dist. S/A

Rua Teodoro da Silva, 907

Grajaú - RJ - 206563-900

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

Carmelita Mulin
www.devmedia.com.br/central/default.asp
(21) 3382-5025

Kaline Dolabella
Gerente de Marketing e Atendimento
kalined@terra.com.br
(21) 3382-5025

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site entre em contato com:

Kaline Dolabella
publicidade@devmedia.com.br

Fale com o Editor

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Guinther Pauli - Editor da Revista
guinther@devmedia.com.br

EDITORIAL

O desenvolvimento multicamadas sempre foi um assunto em alta aqui na ClubeDelphi. De fato, fazem mais de 80 edições que abordamos temas como DataSnap / MIDAS, ClientDataSet, Multitier etc. A possibilidade de dividir uma aplicação client/server em mais uma camada, tornando a solução 3 camadas, parece algo distante. Penso que os desenvolvedores não enxergam os reais benefícios que essa arquitetura proporciona. E com o Delphi é fácil!

O objetivo da matéria em destaque desta edição não é reforçar tudo o que já foi falado nesse mundo de informações, mas mostrar uma abordagem alternativa, mais simples, para o processo de migração. Ao mesmo tempo que desvenda detalhes importantíssimos da arquitetura DataSnap, como o funcionamento da comunicação multicamadas com IAppServer e como funciona o dbExpress na obtenção e resolução de DataPackets. É sem dúvida um estudo profundo nessa poderosa arquitetura, que vai fazer você analisar com outros olhos o DataSnap.

Detalhes como COM+, dbExpress, DataSnap, LocalConnection, ObjectPooling, SimpleObjectBroker, DataSetFields, IAppServer, rodar a aplicação servidora como biblioteca, depuração do servidor, são detalhes que são revelados por completo nesse também completo artigo. Sem dúvida, o artigo mais completo sobre DataSnap que já publicamos, então, não perca mais tempo, migre suas aplicações agora mesmo sem medo e dê um adeus ao mundo client/server e boas-vindas a uma arquitetura muito mais robusta, escalável, profissional e fácil de manter e distribuir.

Ainda nesta edição, três artigos Web. Dois sobre o novíssimo suporte ao ASP.NET 2.0 no Rad Studio 2007. Aprenda o que são, para que servem e como construir portais personalizáveis com Web Parts. Quem conhece o iGoogle vai se sentir familiarizado com a tecnologia. Você pode ter vários "mini-aplicativos" rodando em uma mesma página, que pode ser totalmente personalizada pelo usuário. Ainda sobre ASP.NET 2.0, veja como trabalhar com o interessante recurso de Themes e Skins, que é um recurso semelhante aos temas do Windows, porém aplicados a uma página Web. No artigo de PHP do Fabrício, aprenda a trabalhar diretamente com os principais bancos do mercado usados em conjunto com o PHP, o Firebird e o MySQL.

Também inauguramos nessa edição a nossa sessão Boas Práticas, que vai trazer dicas de como construir aplicações com mais qualidade. Além do artigo de boas práticas multicamadas já comentado, começando em grande estilo temos um artigo de testes unitários do Marco e do Marcelo. Sem dúvida é uma bela prática aplicar testes no seu código para garantir a qualidade, e o Rad Studio pode ajudar muito nesse sentido. O Luciano fala sobre reaproveitamento de código, trazendo dicas e pequenos trechos de código, tanto para Win32 e ASP.NET, que podem facilmente serem utilizados em vários aplicativos, sem redundância. E finalmente, o Paulo Quicoli tem uma boa idéia e também uma boa prática que é notificar de forma elegante o usuário sobre erros da aplicação, ao mesmo tempo que também constrói um elegante framework para isso.

Guinther Pauli

guinther@devmedia.com.br
Microsoft Certified: MCP, MCAD, MCSD.NET, MCTS, MCPD
Delphi Certified: 6, 7, 2005, 2006, Web, Kylix

ClubeDelphi PLUS

DevMedia group

A revista ClubeDelphi é parte integrante da assinatura ClubeDelphi PLUS.
Para mais informações sobre o pacote PLUS, acesse:
http://www.devmedia.com.br/clubedelphi/portal.asp

Hospede com quem fala sua língua!

Local host

Hospedagem Profissional
Revenda
Parcerias

Ativação imediata!

Faça sua assinatura, tenha 30 dias grátis

e seu primeiro boleto para 60 dias após assinatura em qualquer plano

UNIX I

a partir R$ 28,00

- 1 GB Espaço em disco
- 60 GB Transferência
- 30 Contas de e-mail
- 10 Domínios
- Painel de Controle em Português
- Atendimento exclusivo

Contratar

UNIX II

a partir R$ 58,00

- 3 GB Espaço em disco
- 120 GB Transferência
- 60 Contas de e-mail
- 15 Domínios
- Painel de Controle em Português
- Atendimento exclusivo

Contratar

UNIX III

a partir R$ 98,00

- 6 GB Espaço em disco
- 300 GB Transferência
- 90 Contas de e-mail
- 20 Domínios
- Painel de Controle em Português
- Atendimento exclusivo

Contratar

R$ 5,90 — **Mini html**
Tráfego 1 GB
Espaço 100 MB
3 C.Ppostal, Espaço 3G
Domínio 3

R$ 15,90 — **html**
Tráfego 25 GB
Espaço 500 MB
20 C.Ppostal, Espaço 20G
Domínio Ilimitado

R$ 15,90 — **ASP**
Tráfego 25 GB
Espaço 500 MB
20 C.Ppostal, Espaço 20G
Domínio Ilimitado
MySql 50m

R$ 20,90 — **PHP**
Tráfego 25 GB
Espaço 500 MB
20 C.Ppostal, Espaço 20G
Domínio Ilimitado
MySql 50m

Formato exclusivo de trabalho para empresas de desenvolvimento de sites, websites, portais, serviços e negócios que necessitam de grandes tráfegos, monitoramento e backup.

www.lhost.com.br

assinar@lhost.com.br

Assinatura
ClubeDelphi PLUS
Mais conteúdo Delphi por menos

Informativo ClubeDelphi

Portal ClubeDelphi

www.clubedelphi.net/portal

+600 vídeo aulas e 7 cursos online

Brinde na web desta edição

2 Vídeos

Como registrar em que folha foi impresso determinado registro
http://www.devmedia.com.br/articles/viewcomp.asp?comp=10007

ASP.NET 2.0 - Stored Procedures
http://www.devmedia.com.br/articles/viewcomp.asp?comp=9982

Para acessar os vídeos utilize os seguintes dados: Login: DVM.PL Senha: 6DOIT

Gostou das vídeo aulas? O portal *www.devmedia.com.br* possui mais de **2 mil vídeo aulas** e **dezenas de cursos online** sobre desenvolvimento de software! Agora você pode comprar as vídeo aulas que preferir e fazer sua própria combinação de vídeos! Saiba mais em *www.devmedia.com.br/creditos*

Edições Anteriores da ClubeDelphi

Você pode comprar todas as edições anteriores da ClubeDelphi através do site DevMedia!

Para isso basta acessar *https://seguro.devmedia.com.br/edicoes_anteriores.asp*

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link: *www.devmedia.com.br/clubedelphi/feedback*

Para votar, você vai precisar do código e senha de banca desta edição, que é:

Login: **DVM.PL** Senha: **6DOIT**

Você não está mais sozinho.

A partir de agora você pode contar com a ajuda da

Chegou a Consultoria On-line DevMedia

Consultoria Técnica + Professor Virtual + Certificação

Mais Informações:
www.devmedia.com.br/consultoria_online 21 3382-5025

DevMedia em seus projetos e estudos.

A DevMedia possui um numeroso time de autores, editores e professores que juntos produzem o material que você está acostumado a encontrar em nosso site e revistas. E são exatamente esses mesmos profissionais que estarão a sua disposição para tirar suas dúvidas e ajudá-lo em seus projetos e estudos. Através de uma plataforma 100% web a Consultoria DevMedia garante sigilo absoluto, eficiência e rapidez em todas as respostas. Finalmente você terá ao seu alcance uma consultoria de qualidade por um preço muito acessível. Consulte nossos planos.

Mais um serviço DevMedia group

Nesta seção você encontra artigos intermediários sobre Delphi Win32 e Delphi for .NET

Perguntas e Respostas

Lendo uma planilha do Excel e gravando em um TStringGrid

Pouco antes do fechamento dessa edição, recebi um e-mail do leitor Lauro Henrique solicitando uma ajuda para fazer a cópia de uma planilha Excel para o banco de dados dele passando por um *StringGrid*. Em uma busca rápida na internet descobri uma pequena função que se baseia nos componentes da paleta *Servers*. Testei pessoalmente a solução e gostei muito do resultado, por isso resolvi passar a bola para frente. Vamos lá:

Desenhe uma tela semelhante à **Figura 1**. Observe que temos um *TStringGrid* da paleta *Additional* e um *Button* da *Standard*. Crie também uma planilha Excel semelhante à **Figura 2**.

Em seguida declare a *Unit ComObj* no *Uses* do formulário e crie uma função com a seguinte declaração:

```
function Xls_To_StringGrid(AGrid:
  TStringGrid; AXLSFile: string): Boolean;
```

Por fim codifique-a como na **Listagem 1**.

Eu fiz algumas alterações básicas no código que faz uso da função criada. No *OnClick* do botão digite o código da **Listagem 2**.

Execute a aplicação e veja o resultado na **Figura 3**. ●

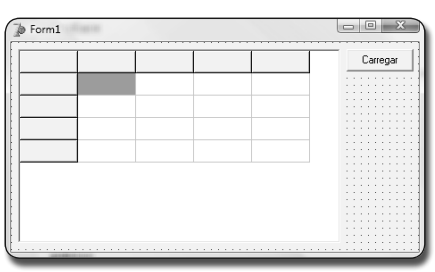


Figura 1. Layout exemplo para aplicação testes

	A	B	C
1	Coluna1	Coluna2	Coluna3
2	Adriano	31	Masculino
3	Guinther	30	Masculino
4	Maria	25	Feminino
5			

Figura 2. Exemplo de planilha

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Listagem 1. Código da função de importação

```
function TForm1.Xls_To_StringGrid(AGrid: TStringGrid;
  AXLSFile: string): Boolean;
const
  xlCellTypeLastCell = $0000000B;
var
  XLApp, Sheet: OLEVariant;
  RangeMatrix: Variant;
  x, y, k, r: Integer;
begin
  Result := False;
  XLApp := CreateOleObject('Excel.Application');
  try
    XLApp.Visible := False;
    XLApp.Workbooks.Open(AXLSFile);
    Sheet := XLApp.Workbooks[ExtractFileName(AXLSFile)].WorkSheets[1];
    Sheet.Cells.SpecialCells(xlCellTypeLastCell, EmptyParam).Activate;
    x := XLApp.ActiveCell.Row;
    y := XLApp.ActiveCell.Column;
    AGrid.RowCount := x;
    AGrid.ColCount := y;
    RangeMatrix := XLApp.Range['A1', XLApp.Cells.Item[X, Y]].Value;
    k := 1;
    repeat
      for r := 1 to y do
        AGrid.Cells[(r - 1), (k - 1)] := RangeMatrix[K, R];
      Inc(k, 1);
      AGrid.RowCount := k + 1;
    until k > x;
    RangeMatrix := Unassigned;
  finally
    if not VarIsEmpty(XLApp) then
    begin
      XLApp.Quit;
      XLAPP := Unassigned;
      Sheet := Unassigned;
      Result := True;
    end;
  end;
end;
```

Listagem 2. Código de chamada da função

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with TOpenDialog.Create(Self) do
  begin
    Filter := 'Arquivos Excel(*.XLS)|*.xls';
    InitialDir := 'C:\';
    if Execute then
      if Xls_To_StringGrid(StringGrid1, FileName) then
        ShowMessage('Arquivo Importado com sucesso!');
  end;
end;
```

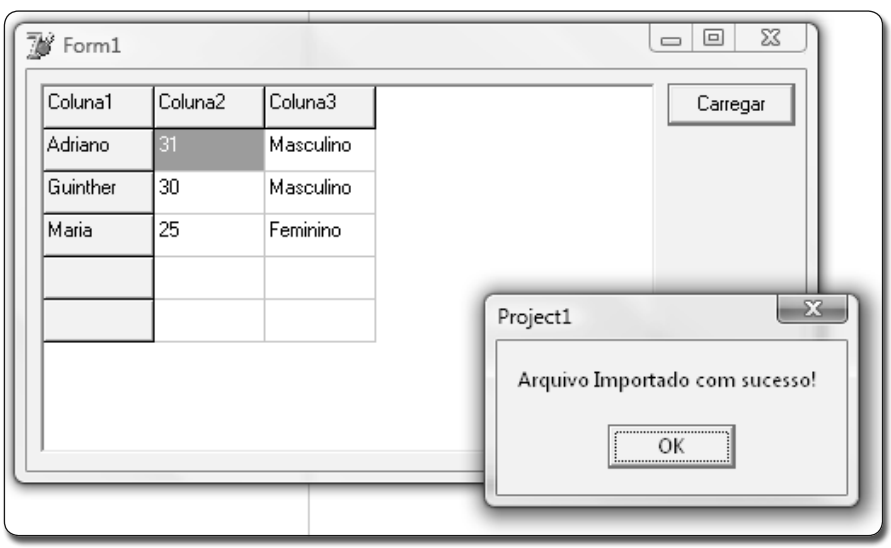
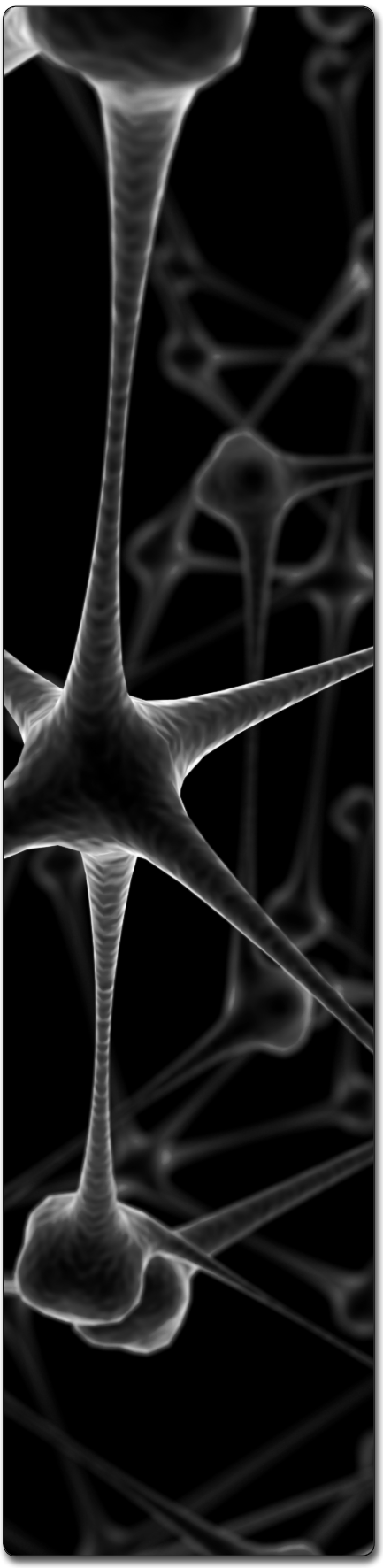


Figura 3. Resultado da importação dos dados da planilha Excel



Nesta seção você encontra artigos intermediários sobre Delphi Win32 e Delphi for .NET

WebParts

Portais e páginas customizáveis com ASP.NET 2.0

Resumo DevMan

A Web tem se tornado um grande canal de comunicação como já mencionamos diversas vezes. Mas além de comunicação, a internet funciona também como uma central de trabalho, entretenimento, informação etc. O acesso a internet cada vez mais freqüente e fácil na vida de todos tem sido o carro chefe que empurra para frente a evolução dos sites. Hoje em dia um site não basta ser bonito e funcional, precisa ser atraente e em alguns casos customizável, personalizável, ou seja, deve permitir que usuário final possa mover e adequar o conteúdo de acordo com sua necessidade. Os WebParts fazem isso e muito mais. É possível movermos de lugar partes de um determinado site e mantê-lo assim para uma futura consulta. É isso que veremos nesse artigo.

Nesse artigo veremos

- Criação de portais e páginas customizáveis;
- Como usar WebParts;
- Usando zonas, catálogos e menus em WebParts.

Qual a finalidade

- Com WebParts podemos criar áreas no Web Site que pode ser personalizadas pelo usuário final.

Quais situações utilizam esses recursos?

- Podemos proporcionar ao usuário a possibilidade de adequar o conteúdo da página de acordo com sua necessidade e gosto.

Guinther Pauli
(guinther@devmedia.com.br)
É Bacharel em Sistemas de Informação, Microsoft Certified: MCP, MCAD, MCSD.NET, Borland Certified: Delphi 6, 7, 2005, 2006, Web e Kylix. Editor Geral das Revistas .net Magazine, ClubeDelphi, WebMobile (.NET) e Mr.Bool. Pode ser contatado pelo endereço guinther@devmedia.com.br

Adriano Santos
(falecom@adrianosantos.pro.br)
é desenvolvedor Delphi desde 1998. Professor e programador PHP. Bacharel em Comunicação Social pela Universidade Cruzeiro do Sul, SP. É Editor Técnico, Colunista e Membro da Comissão Editorial das revistas ClubeDelphi. Gerente de Desenvolvimento da SoftPark, empresa parceira Borland. Mantém o blog Delphi to Delphi (www.delphitodelphi.blogspot.com) com dicas, informações e tudo sobre desenvolvimento Delphi

Imagine uma interface onde você possa executar várias aplicações simultaneamente. Como se a tela do seu computador fosse dividida em "janelas", e em cada janela você tivesse um programa sendo executado, como uma planilha, um documento, o sistema de controle de estoque etc. Soa familiar? Acho que alguém já teve essa idéia há pelo menos uns 20 anos.

Vamos transportar essa idéia para um ambiente Web, uma página da internet. Pense na possibilidade de rodar várias aplicações em uma mesma página de um site ou portal. A mesma funcionalidade de "janelas", tão banal atualmente nos sistemas operacionais. Imagine essa funcionalidade na Web. Imaginou? É exatamente isso o que as *WebParts* do *ASP.NET 2.0* fazem. Uma página Web dividida em janelas, onde em cada janela você roda a aplicação de sua escolha,

com funcionalidades de abrir, mover, minimizar, fechar etc.

Este é o tema que será abordado por este artigo. Iremos aqui, explicar esse conceito e demonstrar de forma prática como implementar *WebParts* em suas aplicações *ASP.NET 2.0*.

Controles WebParts

O conceito de portais não é algo novo e foi agora incorporado à versão 2.0 do *ASP.NET*. É importante ressaltar que *WebPart* não é um único controle no *ASP.NET*, e sim uma nova forma de se implementar aplicações Web. Para implementar *WebParts* no *ASP.NET*, temos um grande conjunto de controles disponíveis. Podemos encontrá-los na paleta *WebParts* no *RAD Studio 2007* (**Figura 1**).

Primeiro exemplo

Vejamos agora um simples exemplo de como implementar as funcionalidades de *WebParts*. Diferente de um ambiente *Desktop*, onde você pode colocar os seus aplicativos em qualquer lugar da sua tela, em um ambiente de *WebParts* você define os locais onde as *aplicações* poderão ser executadas. Estes espaços são chamados de *zonas*. Pense em uma tabela *HTML* com diversas células, em que você pode arrastar *mini-aplicativos* entre esses espaços.

Vamos criar um exemplo para entendermos melhor essa característica. Crie um novo projeto *ASP.NET* no *RAD Studio 2007* usando o menu *File>New>ASP.NET Web Applications – Delphi for .NET* e nomeia como *WebParts*. Insira o controle *WebPartManager* na página *default*. Esse controle é que irá *gerenciar* as características das *WebParts*. Abaixo do *WebPartManager*, insira uma tabela com duas linhas e duas colunas. E em cada célula da sua tabela insira um controle *WebPartZone*. Sua página deve se parecer com o demonstrado na **Figura 2**.

No design da sua página, no *RAD Studio 2007*, as zonas se comportam como *containeres* onde podemos incluir controles dentro delas. Arraste um controle de calendário na primeira zona. Na zona 2 arraste um controle *Button* e um *TextBox*. Veja que você pode colocar mais de um controle em uma mesma zona. O nosso exemplo deve ficar como o da **Figura 3.**

No *Page_Load* da página *default*, inclua o código a seguir. O código define a propriedade *DisplayMode* do *WebPartManager* para o modo *design*. Em seguida execute o seu projeto. Sua página deverá se parecer com o demonstrado na **Figura 4**.

```
WebPartManager1.DisplayMode :=
WebPartManager1.DesignDisplayMode;
```

Observe que cada controle inserido nas zonas é *revestido* de uma janela, contendo uma barra de título, uma borda e um menu, identificado pela pequena seta no canto direito. Se passarmos o mouse sobre o título da janela (por enquanto todos estão com *Untitled*) veremos que o ponteiro do mouse modificará sua forma. Isso nos indica que podemos clicar e arrastar essa janela entre as zonas! Faça testes e mova os controles entre as zonas, veja como a dinâmica é bem parecida com a de janelas do Windows. Verifique na **Figura 5** como você pode mover as *WebParts* entre as zonas.

Características de uma WebPart

Quando incluímos controles nas zonas, eles aparecem com algumas características peculiares. Essas características lhes dão um aspecto de "janela", com uma barra de título, uma borda e um menu. Essas características definem o aspecto visual de uma *WebPart* e são implementadas em qualquer controle que for incluído dentro de uma *WebPartZone*.

Podemos alterar o *layout* gráfico das *WebParts* aplicando um dos formatos *default* que temos no *RAD Studio 2007*, ou, simplesmente, trabalhando com os estilos do controle. Para aplicarmos um formato diferente à *WebPart*, basta acessar a opção *AutoFormat* da *WebPartZone*, através das suas *Tasks*. Veja um exemplo na **Figura 6.**

Observe que fazemos a alteração de *layout* na zona e não diretamente no controle. As características gráficas das *WebParts* que "revestem" os nossos controles serão herdadas das zonas em que os controles se encontram.

Umas das características mais impor-

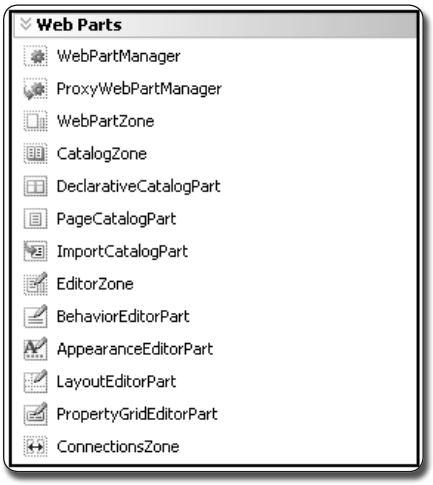


Figura 1. Controles de WebParts na ToolBox do RAD Studio 2007

Figura 2. Página ASP.NET com quatro zonas

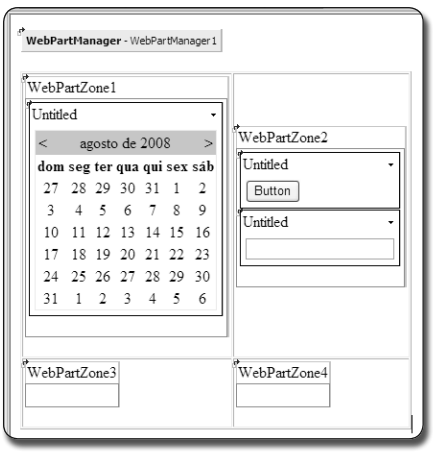


Figura 3. Controles dentro das zonas

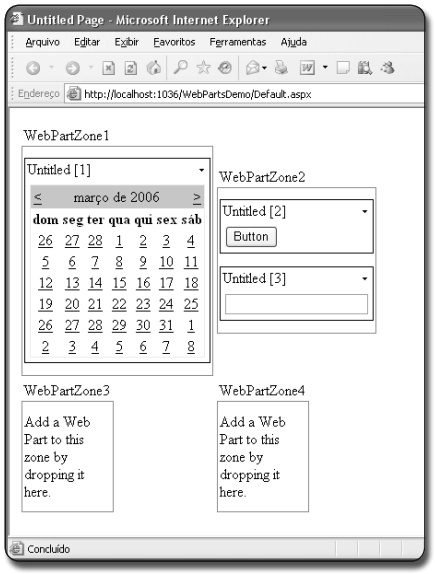


Figura 4. Projeto com WebParts

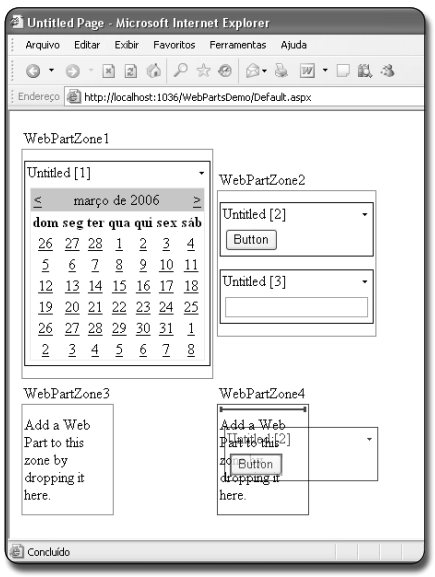


Figura 5. Movendo o controle Button da zona 2 para a zona 4

tantes da *WebPart* é o menu que encontramos no canto superior direito. Através desse menu podemos minimizar, restaurar e fechar a *WebPart*. Veja a **Figura 7**. As opções que nos aparecem por *default* são as de minimizar e fechar. Ao minimizarmos uma *WebPart*, todo o seu conteúdo é "escondido", ficando apenas a barra de título aparecendo, juntamente com o menu de opções. Quando minimizada, as opções que temos no menu são as de restaurar e fechar. Ao restaurarmos, a *WebPart* volta a ter o seu aspecto original. Se fecharmos uma *WebPart* ela passa a não aparecer mais em nossa página.

Catálogo de WebParts

Para onde vai uma *WebPart* quando a fechamos? E como fazer para recuperá-la em nossa página? Todas as *WebParts* com que estamos trabalhando são controles que nós mesmos incluímos nas *WebPartZones*. Todas essas *WebParts* fazem parte do que chamamos de catálogo de *WebParts* da página.

Para visualizarmos as *WebParts* que foram fechadas pelo usuário, precisamos de mais dois controles em nossa página. Primeiro incluímos um controle *CatalogZone*. Esse controle é como uma *WebPartZone*, onde podemos incluir outros controles dentro dele. Devemos então incluir um controle *PageCatalogPart* dentro do *CatalogZone* que acabamos de inserir. A sua página deve se parecer com a da **Figura 8**.

No começo do nosso exemplo incluímos um código no *Page_Load* da página. Nesse código, configuramos a propriedade *DisplayMode* do nosso *WebPartManager* com a constante *DesignDisplayMode*. Para podermos acessar o nosso catálogo de *WebParts*, precisamos trocar a configuração modificando essa propriedade com a constante *CatalogDisplayMode*. Isso define o "modo" em que

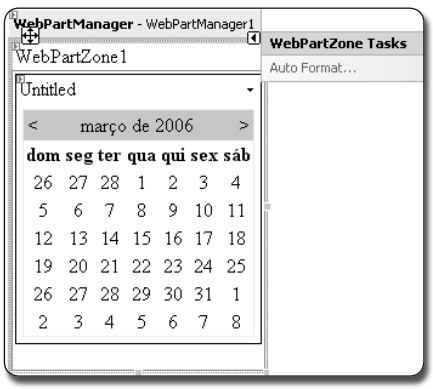


Figura 6. Aplicando um formato às WebParts da zona

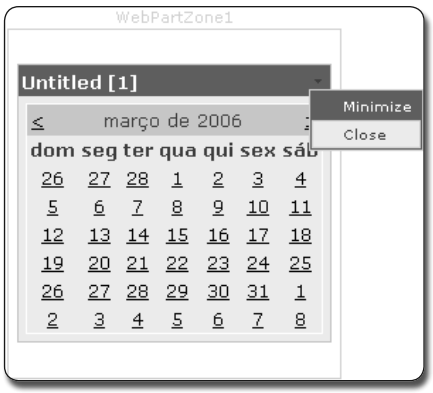


Figura 7. Menu da WebPart

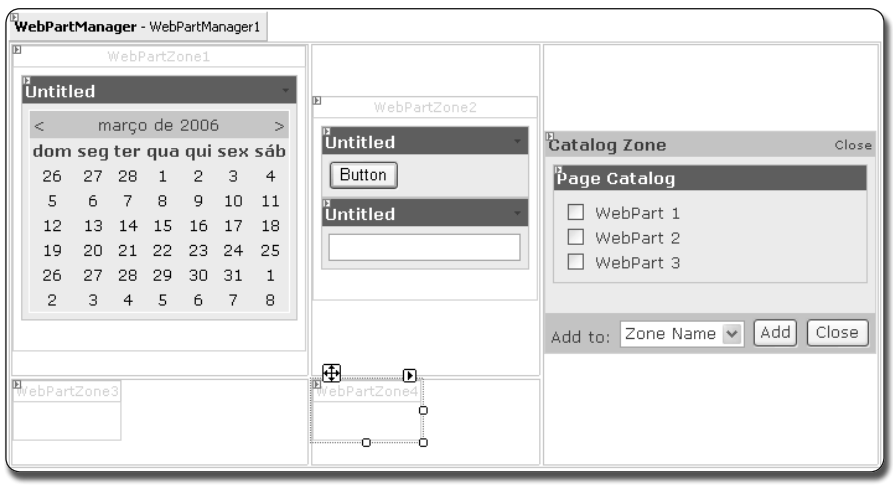


Figura 8. Controles CatalogZone e PageCatalogPart

estamos trabalhando com as *WebParts*. O *Page_Load* deve ficar como a seguir:

```
WebPartManager1.DisplayMode :=
WebPartManager.CatalogDisplayMode;
```

Execute o seu projeto e faça um teste. Veja na **Figura 9,** que o calendário está fechado e não aparece mais na página. Ele aparece dentro do *Page Catalog*, que é o catálogo de *WebParts* da nossa página. Nele irão sempre aparecer as *WebParts* fechadas pelo usuário. Para devolver a *WebPart* para a página, devemos selecioná-la marcando o seu respectivo *checkbox*, escolher a zona em que vamos adicioná-la e clicar no botão *Add*.

Editando uma WebPart em tempo de execução

Uma característica que podemos observar é que todas as *WebParts* estão com o nome *Untitled*. Esse título não pode ser alterado nas propriedades do controle. Então, como fazermos para alterá-lo? Assim como o título da *WebPart*, temos um conjunto de propriedades que podem ser alteradas em modo de execução do projeto. Para fazermos isso, precisaremos novamente incluir mais dois controles em nossa página. Primeiro incluímos o controle *EditorZone*, e em seguida incluímos dentro do *EditorZone* o controle de edição que precisamos.

Observe que temos alguns controles que terminam com o nome *EditorPart*. Todos esses controles servem para editarmos características das *WebParts* da nossa página. Para o nosso exemplo vamos incluir um controle *AppearanceEditorPart* dentro do controle *EditorZone*. Sua página deve se parecer com a da **Figura 10**.

Veja que no controle *AppearanceEditor Part* já aparecem as propriedades de

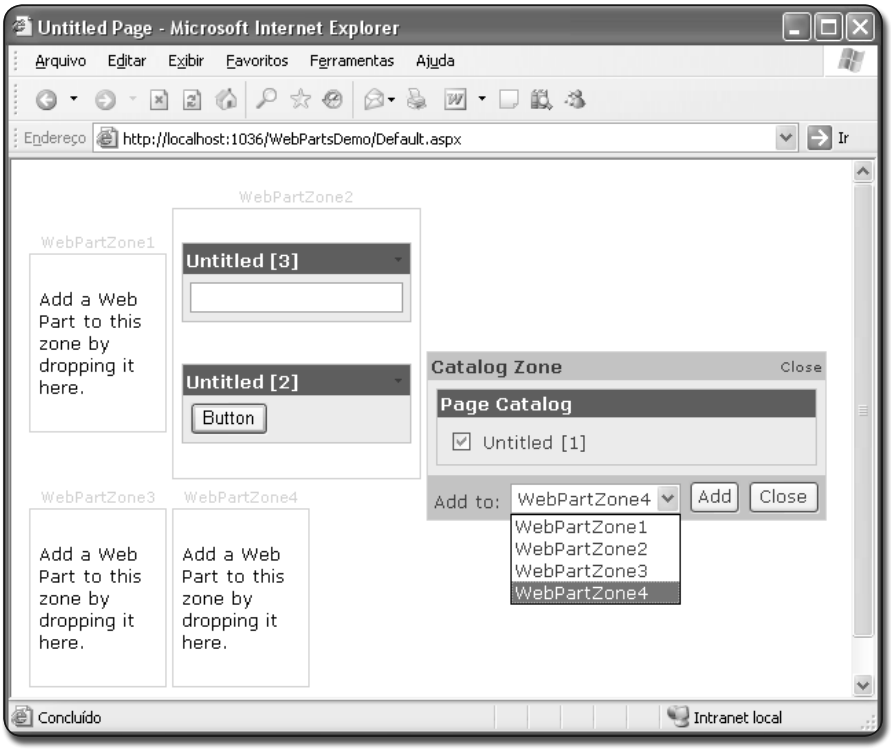


Figura 9. Acessando o catálogo de WebParts da página

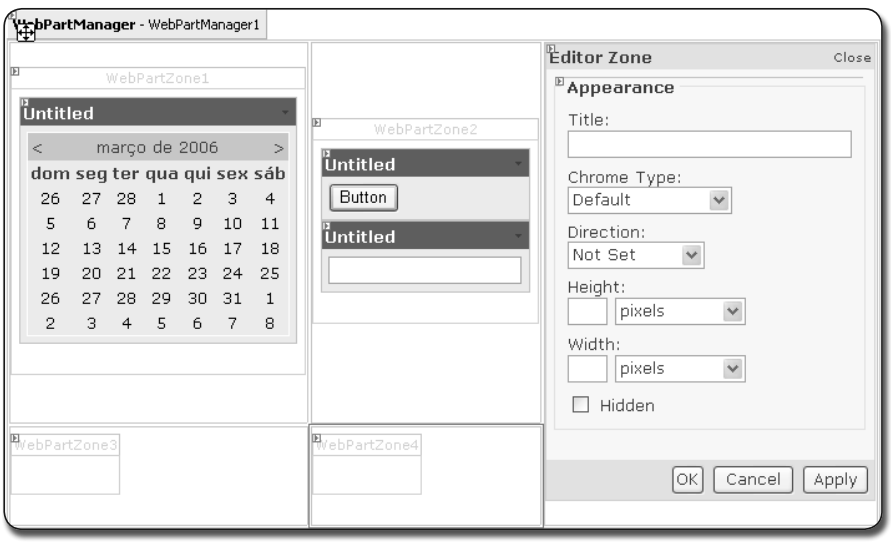


Figura 10. Incluindo controles para edição de propriedades das WebParts

aparência que poderemos alterar nas *WebParts*, inclusive a propriedade *Title*. Para completar, precisamos novamente alterar o *DisplayMode* do *WebPartManager* para a constante *EditDisplayMode*. Como explicamos anteriormente, o *DisplayMode* define o "modo" que estamos trabalhando com as *WebParts*. Para podermos editar as propriedades da mesma, o *DisplayMode* precisa ser modificado para *EditDisplayMode*. Veja como deve ficar o seu evento *Page_Load:*

```
WebPartManager1.DisplayMode =
   WebPartManager.EditDisplayMode;
```

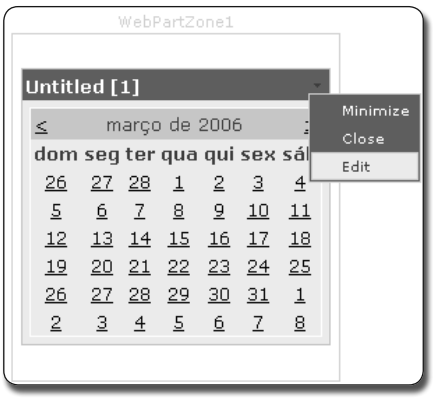


Figura 11. Opção Edit do menu da WebPart

Depois de feita a alteração, execute o seu projeto. A princípio não houve nenhuma modificação, mas se clicarmos na seta de menu de um dos controles, veremos que uma opção foi adicionada aos menus das *WebParts*. É a opção *Edit*, veja a **Figura 11.** Clique na opção *Edit* e veja que o controle de edição da *WebPart* irá aparecer, como mostra a **Figura 12**. Todas as alterações feitas aqui valem para a *WebPart* selecionada. Ao clicar em OK, o controle de edição é fechado.

Conclusão

Como vimos, os controles para a implementação de *WebParts* são bem simples de serem utilizados. Em todos os testes que fizemos aqui, apenas uma linha de código foi utilizada, sendo que toda a implementação foi feita apenas com a inclusão dos controles nas páginas. Nos testes que realizamos, foram incluídos apenas controles simples nas zonas. Em uma aplicação real, devemos criar nossos *mini-aplicativos* como *User Controls*, para que sejam exibidos nas zonas e funcionem como *WebParts*. Há também a possibilidade de criarmos controles que herdam da classe *WebPart*. ●

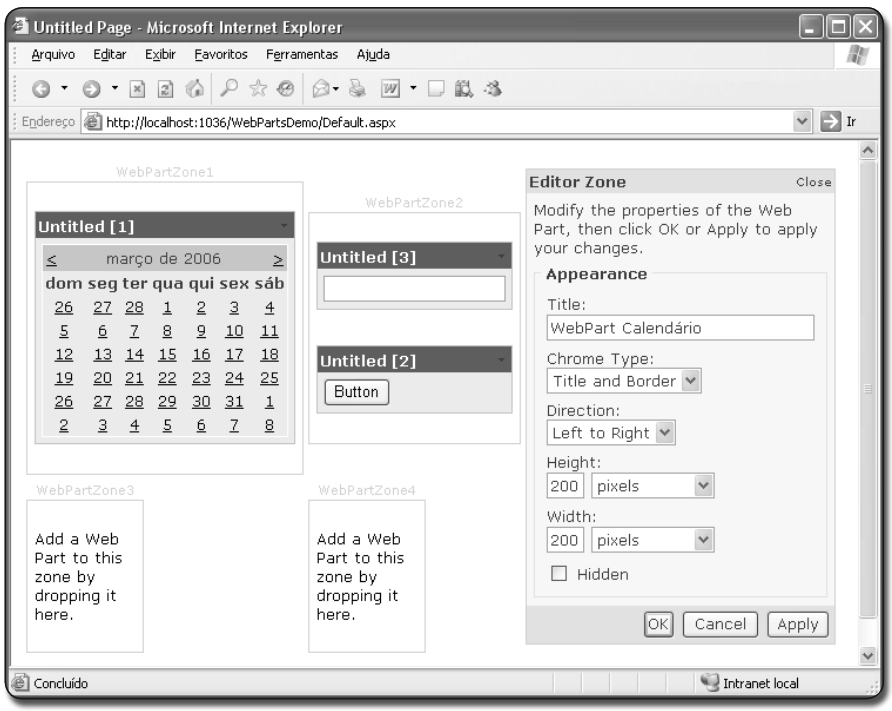
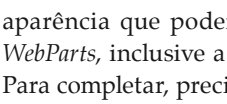


Figura 12. Editando propriedades de aparência da WebPart



Nota do DevMan

Talvez você não tenha percebido, mas todos os labels nos painéis do WebParts estão em inglês. Isso pode facilmente ser contornado verificando as propriedades dos controles. Por exemplo: o controle *WebPartZone* permite alterar a propriedade *EmptyZoneText* que exibe uma mensagem padrão quando não há nenhum controle na zona.

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

10 Motivos
para hospedar seu site na RedeHost!

30 dias GRÁTIS para testar.

Mais de 15 mil clientes e 6 anos no mercado.

Até 50Gb de espaço para e-mails ou site.

Planos a partir de R$ 13,90.

Webmail em Ajax com calendário, tarefas, anotações e RSS.

ASP, PHP, .net e MySQL no mesmo plano.

Domínios adicionais ilimitados (apontam para o site principal).

Data center no Brasil com Links Redundantes.

Infra-estrutura com servidores Dell e redes Cisco.

Estatísticas de acesso com 75 relatórios.

RedeHost

www.redehost.com.br

Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software

Testes unitários de software

Aprenda a efetuar testes unitários no Delphi 2007 com nUnit e csUnit

Resumo DevMan

Entre testar manualmente um sistema e testar profissionalmente, certamente fico com a melhor delas: a segunda opção. A qualidade de software é hoje uma das maiores preocupações de qualquer empresa de tecnologia envolvida no desenvolvimento de sistemas. Por isso veremos nesse artigo como criar testes unitários capazes de identificar possíveis erros em rotinas e classes.

Utilizaremos os programas nUnit e csUnit para realizar tais tarefas e chegaremos a conclusão que não é tão difícil manter o sistema bem estruturado e funcionando perfeitamente.

Nesse artigo veremos

• Testes unitários caixa branca;
• Criação de rotinas para testes;

Qual a finalidade

• Preparar o sistema para efetuar testes em classes do sistema, evitando maiores problemas no futuro;

Quais situações utilizam esses recursos?

• Os recursos vistos aqui se enquadram bem em aplicações Orientadas a Objetos e mantém as rotinas do sistema concisas e coerentes. Em qualquer sistema podemos aplicar essas técnicas;

Marcelo Santos Daibert
(marcelo@daibert.net)
é professor do Curso de Bacharelado em Ciência da Computação da FAGOC - Faculdade Governador Ozanam Coelho na graduação e pós-graduação (especialização), Mestrando e Especialista em Ciência da Computação pela Universidade Federal de Viçosa e Bacharel em Sistemas de Informação pela Faculdade Metodista Granbery. Gerente técnico da Optical Soluções em Informática.

Marco Antônio Pereira Araújo
(maraujo@devmedia.com.br)
é professor dos Cursos de Bacharelado em Sistemas de Informação do Centro de Ensino Superior de Juiz de Fora e da Faculdade Metodista Granbery, Doutorando e Mestre em Engenharia de Sistemas e Computação pela COPPE/UFRJ, Especialista em Métodos Estatísticos Computacionais e Bacharel em Informática pela UFJF, Analista de Sistemas da Prefeitura de Juiz de Fora, Editor da Engenharia de Software Magazine.

Principalmente nos últimos anos, foi possível observar um grande avanço em todo o processo de desenvolvimento de software, motivado principalmente pelas necessidades impostas por um mercado cada vez mais competitivo. Novas técnicas de desenvolvimento de software, novos métodos, formas de documentação, linguagens, entre outros, foram desenvolvidos. A necessidade de aspectos de usabilidade nos sistemas começaram a fazer diferença e a palavra *qualidade* cada vez mais foi se destacando nas pautas dos desenvolvedores. A qualidade no software passou a ser uma busca cada vez mais freqüente pelos desenvolvedores e empresas de software.

Uma falha no software pode ocasionar perdas financeiras, perdas de informações, que muitas vezes é até mais catastrófica do que perdas financeiras, e, dependendo da criticidade da aplicação, até perda de vidas humanas.

Neste contexto, um dos temas importantes no processo de desenvolvimento de software é a fase de testes das aplicações, onde a mesma é submetida a uma carga de testes a fim de identificar falhas antes mesmo do software entrar no ambiente de produção.

Existem algumas estratégias de teste de software e a principal utilizada atualmente em muitas empresas de desenvolvimento, principalmente as de médio e pequeno porte, é a utilização dos testes manuais. Nesta abordagem é atribuída a função de testar as aplicações desenvolvidas a um ou mais membros da equipe de forma manual, onde o sistema é executado e as informações são inseridas manualmente, verificando se os resultados esperados foram alcançados. No entanto, esta abordagem é muito ineficaz e demorada. As melhores técnicas de teste são baseadas em algum processo automatizado, onde é possível executar uma maior quantidade de testes, buscando assim testar o máximo possível dos requisitos de um software. Nesta abordagem, existem algumas estratégias, entre elas: testes unitários (o que este artigo aborda), testes funcionais, teste de desempenho, testes em banco de dados, testes em *WebServices*, entre outros.

O teste unitário, um tipo de teste caixa branca, por ser baseado na estrutura lógica do código, é responsável por testar a unidade de codificação da aplicação. Em um sistema orientado a objetos esta unidade pode ser representada pela própria classe ou pelos métodos das classes. Dada uma entrada, o teste unitário deve aferir o resultado, levando em consideração todos os possíveis caminhos do algoritmo e o seu processamento. Para definir o número mínimo de casos de teste para cobrir as possibilidades de caminhos de processamento de um trecho de código, é apresentada a métrica de software chamada complexidade ciclomática, que define o número de caminhos independentes que um algoritmo deve percorrer para efetuar todos os processamentos.

O objetivo deste artigo é apresentar de forma prática a abordagem de desenvolvimento de software baseada em testes unitários, conceituar a técnica de programação por intenção e exibir a utilização prática da estratégia de teste unitário usando as ferramentas *nUnit* e *csUnit* no ambiente de desenvolvimento *CodeGear RAD Studio 2007* para *.NET*. Tanto o *nUnit*, quanto o *csUnit*, são compatíveis com qualquer linguagem e ambiente de desenvolvimento *.NET*.

Desenvolvimento Baseado em Testes

O *Desenvolvimento Baseado em Testes* (TDD – *Test Driven Development*) é uma abordagem que faz uso das técnicas de teste de software para minimizar a quantidade de falhas na aplicação desenvolvida. Para isso, todo o processo de desenvolvimento de software é baseado em testes, a começar pela técnica de desenvolvimento de teste antes da codificação (TFD – *Test First Development*).

Nesta técnica, o desenvolvedor deve planejar e construir o teste para um trecho ou módulo do sistema que está sendo construído, antes mesmo de sua codificação. Com isso, a qualidade do caso de teste é aperfeiçoada, já que o mesmo é feito de forma incremental, juntamente com a codificação, como pode ser visualizado em um diagrama de atividades na **Figura 1**. Esta técnica faz uso de uma estratégia chamada programação por intenção.

Na **Figura 1**, é possível identificar claramente os passos do *TFD*. A primeira atividade observada é a *Adicionar Teste*. Nela, o teste é escrito e então executado no segundo passo. Certamente o teste irá falhar, já que não existe codificação da função, o que justifica a terceira atividade, chamada *Modificação / Refatoração*. Nesta atividade o desenvolvedor irá codificar a função a fim de fazer o teste ser executado com sucesso. Após, na última atividade, o teste é executado novamente. Se ele passar, o desenvolvimento continua ou finaliza e, caso o teste falhe, há a necessidade de uma nova iteração na atividade *Modificação / Refatoração* até que o teste seja executado com sucesso.

A programação por intenção é uma abordagem de programação que induz a codificação usando classes, métodos ou módulos do sistema que ainda não existem e serão criados futuramente para atender às necessidades de compilação da aplicação. No contexto de desenvolvimento baseado em testes, a programação por intenção auxilia a criação dos casos de testes, quando utilizada a técnica de desenvolvimento de testes antes da codificação, já que o teste produzido afere um método ainda inexistente. E este método, por sua vez, pode fazer uso de recursos da aplicação também inexistentes.

A idéia central desta técnica é a comunicação com as intenções do desenvolvedor ao codificar a aplicação. Mesmo gerando um código fonte não compilável, é possível definir e limitar o escopo de ação de determinado método, além de traçar os passos de desenvolvimento para atender às necessidades de compilação, inclusive utilizando técnicas de refatoração de

Nota do DevMan

Caixa-Preta

Técnica de teste, também chamado de Teste Funcional, em que o componente de software a ser testado é abordado como se fosse uma caixa-preta, ou seja, não se considera o comportamento interno do mesmo. Dados de entrada são fornecidos, o teste é executado e o resultado obtido é comparado a um resultado esperado previamente conhecido.

O componente de software a ser testado pode ser um método, uma função interna, um programa, um componente, um conjunto de programas e/ou componentes ou mesmo uma funcionalidade. A técnica de teste de Caixa-Preta é aplicável a todas as fases de teste - fase de teste de unidade (ou teste unitário), fase de teste de integração, fase de teste de sistema e fase de teste de aceitação.

A aplicação de técnicas de teste leva o testador a produzir um conjunto de casos de teste (ou situações de teste). A aplicação combinada de outra técnica - Técnica de Particionamento de Equivalência (ou uso de Classes de Equivalência) permite avaliar se a quantidade de casos de teste produzida é coerente. A partir das classes de equivalência identificadas, o testador irá construir casos de teste que atuem nos limites superiores e inferiores destas classes, de forma que um número mínimo de casos de teste permita a maior cobertura de teste possível

código-fonte. Para isso, muitas vezes é necessário utilizar os chamados objetos *Mock*, com o objetivo de simplificar a utilização da programação por intenção e a criação dos casos de teste. Objetos *Mock* são objetos falsos, ou de fachada, com o objetivo de substituir recursos não disponíveis ou inadequados, possibilitando a criação dos casos de teste e execução de testes unitários no sistema.

Independente da utilização desta abordagem de desenvolvimento baseado em testes, a utilização das técnicas de teste se configuram como uma importante ferramenta para buscar a qualidade das aplicações e a minimização de defeitos no *software*.

Configuração do ambiente no Delphi 2007

O *nUnit* e o *csUnit* são ferramentas independentes do ambiente de desenvolvimento. Basicamente, ambas carregam uma biblioteca compilada no ambiente *.NET* e executam os testes configurados.

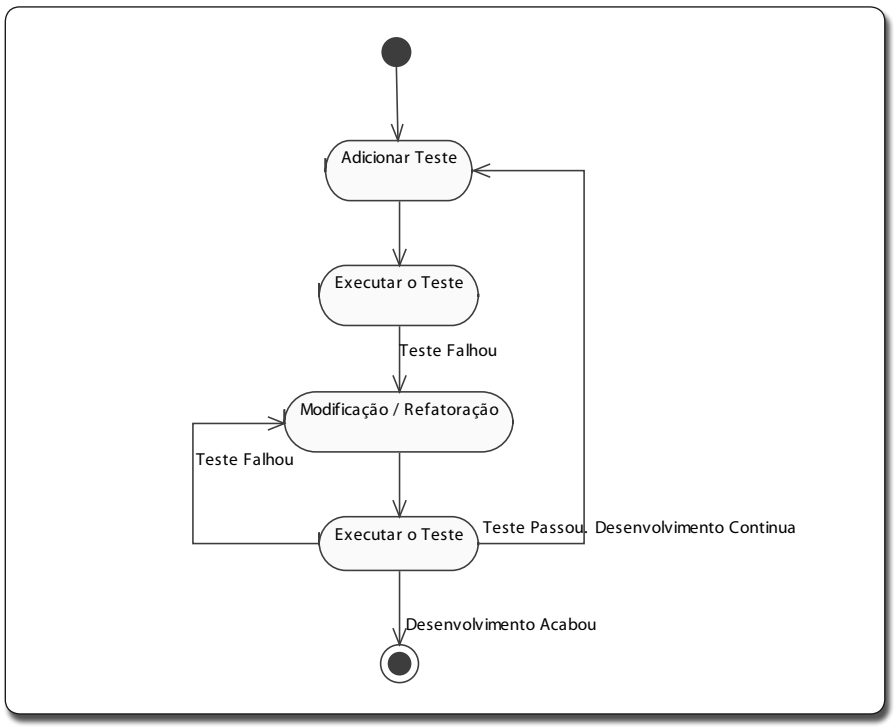


Figura 1. Passos do TFD – Test First Development

Neste contexto, para que seja possível a execução dos testes de forma automatizada pelas ferramentas de teste unitário, é necessário configurar um projeto no Delphi 2007 de forma a gerar uma biblioteca (neste caso, um arquivo *DLL*). Para isso, acesse o *menu* do Delphi e crie um projeto de testes unitário: *File>New>Other>Unit Test>Test Project*, como visualizado na **Figura 2**. Esta opção irá configurar o ambiente do Delphi para a geração de uma biblioteca *DLL* com os testes que serão configurados a seguir. No entanto, esta não é a única alternativa. É possível também selecionar a opção *Delphi for .NET Projects>Library*, como apresentado na **Figura 3**. Nesta opção será configurado um ambiente para a geração de uma DLL, o que cumpre com as necessidades para a geração dos testes e posterior execução automatizada nas ferramentas.

Com estas ações, sempre que o projeto for compilado será gerado um arquivo *.dll* com o mesmo nome do projeto (neste caso, foi dado o nome *libClasses* – o que gera uma *DLL* chamada *libClasses.dll*). É este arquivo que deve ser carregado tanto pelo *nUnit* quanto pelo *csUnit* para a execução dos testes.

Com o projeto criado, é necessário adicionar uma referência a uma biblioteca do *nUnit* para que seja possível utilizar métodos e funcionalidades disponibilizadas pela ferramenta, tornando possível assim a criação dos casos de teste. Na aba *Project Manager,* clique com o botão

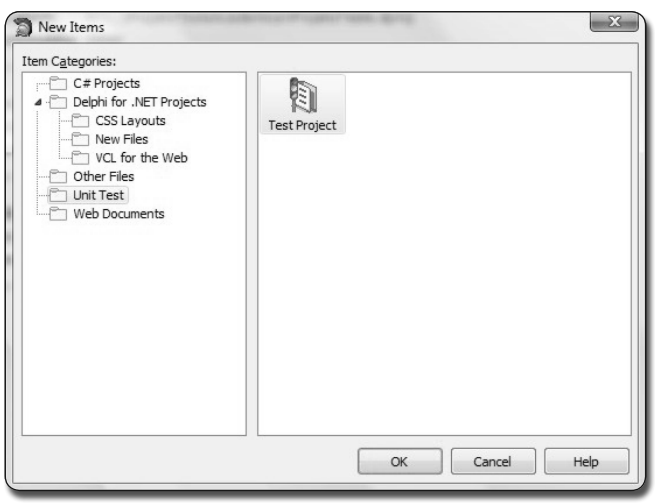


Figura 2. Projeto de Teste no Delphi 2007

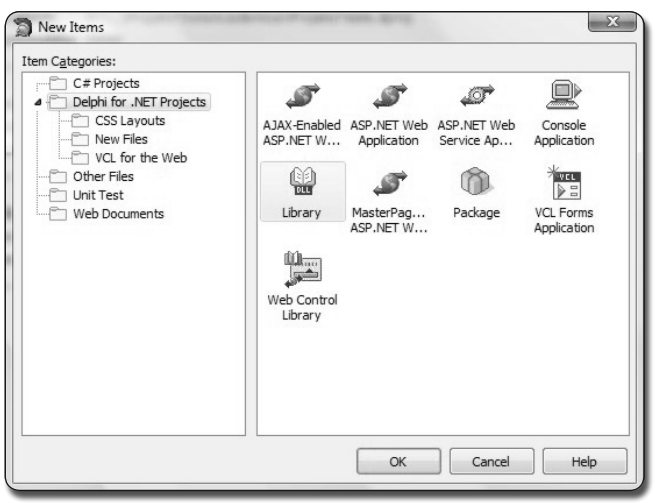


Figura 3. Biblioteca – Library no Delphi 2007

direito no item *References* e escolha a opção *Add Reference*. Ao invocar esta opção, é apresentada uma janela para escolha de referências a serem adicionadas ao projeto. A referência do *nUnit* deve ser encontrada na aba *.NET Assemblies* com o nome *nunit.framework*, como exemplificado na **Figura 4**. Caso não seja possível encontrá-la, é necessário verificar se o *nUnit* está realmente instalado (verifique o local para download da ferramenta na sessão *links* deste artigo) ou adicionar a referência de forma manual.

Para adicionar a referência manual, clique em *Browse* e localize o arquivo da biblioteca no disco. O nome do arquivo é *nunit.framework.dll* e fica armazenado dentro do diretório *bin* do diretório onde o *nUnit* foi instalado. Para finalizar, clique em *Add Reference* e *OK*. Depois de registrada no projeto, a referência da biblioteca do *nUnit* é exibida na seção *References* no *Project Manager*, como apresentado pela **Figura 5**.

É importante destacar que foi adicionado ao projeto somente uma referência à biblioteca do *nUnit*. Os casos de teste construídos irão também rodar no *csUnit*. Isso é possível, pois o *csUnit* é compatível com as diretivas de compilação e atributos do *nUnit*.

Estudo de caso e criação dos testes

Buscando exemplificar o uso das ferramentas de teste unitário, é proposto um estudo de caso criado utilizando o ambiente de desenvolvimento do *CodeGear RAD Studio 2007 for .NET*. Nele é apresentada uma classe chamada *TAluno*, com os atributos *freqüência*, *nota1*, *nota2* e *provafinal*.

Esta classe possui dois métodos: uma *procedure* chamada *setDados*, responsável por atribuir os valores aos atributos da classe, respeitando assim o princípio do *encapsulamento*, e a *function calcularAprovacao*, responsável por verificar se o aluno foi aprovado ou não, de acordo com sua freqüência e notas. Esta classe foi codificada em uma *unit* chamada *Aluno.pas* e está sendo apresentada na **Listagem 1**.

O método *calcularAprovacao* verifica inicialmente se a freqüência do aluno é menor que 75%. Caso seja, o aluno é reprovado de imediato, fazendo com que a função retorne falso. Caso contrário, é verificada a média da nota1 com a nota2. Caso essa média seja menor que 3, o aluno também é reprovado.

Caso contrário, se a média for maior ou igual a 7, o aluno é aprovado. Se a média for maior ou igual a 3 e menor que 7, o aluno estará em prova final e será aprovado caso o valor da média de sua nota de prova final com a média anterior for superior ou igual a 5. Caso contrário, ele será reprovado.

Com a classe *Aluno* definida, o objetivo agora é criar casos de teste para o método *calcularAprovacao*, buscando testar as regras descritas pelo método. Para isso, neste estudo de caso, foi criada uma nova *unit* com o nome *AlunoTeste* e esta é apresentada na **Listagem 2**. É importante observar que esta *unit* faz referência direta à classe *TAluno* (*uses TAluno*) e à classe *nUnit.Framework (uses nUnit.Framework)*, que contém os métodos e atributos disponibilizados pelo *nUnit* para a criação dos casos de teste.

A **Listagem 2** apresenta a codificação da *unit TesteAluno*, contendo uma classe chamada *TAlunoTeste*, sua definição e implementação dos métodos *testeAprovacao1* até *testeAprovacao5*, que são os casos de teste, e que representam o número de caminhos do algoritmo apresentado. Deve-se observar o uso do atributo *[TestFixture]*, antes da definição da classe *TAlunoTeste*, na linha 9. Esse atributo deve ser utilizado antes das classes definidas como classes de teste, para que o *nUnit* possa reconhecê-las dessa forma.

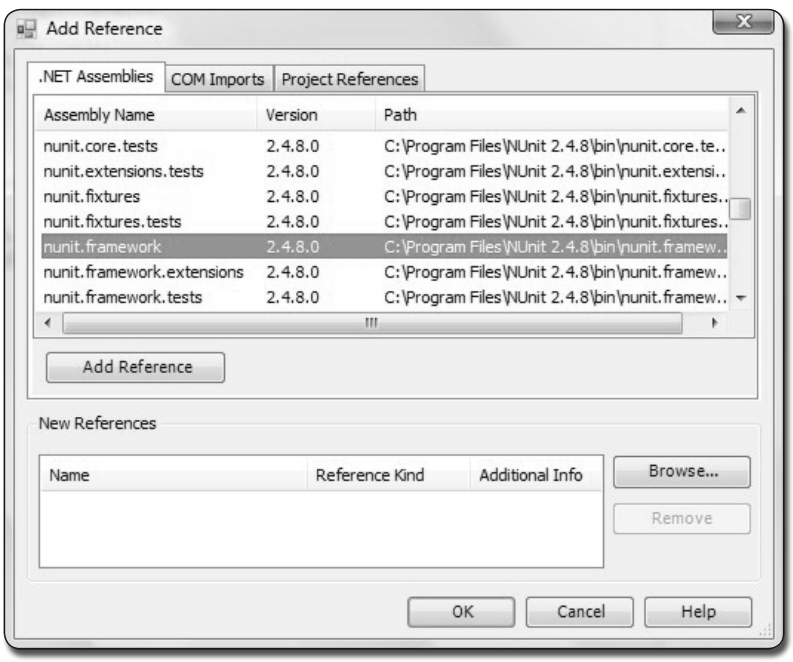


Figura 4. Adicionando a referência ao nUnit

Figura 5. Referência ao nUnit adicionada

O *setUp* é responsável pela inicialização do teste, como a instanciação do objeto, por exemplo. É importante destacar o uso do atributo *[SetUp]* antes da sua definição, como pode ser visto na linha 31. Esse atributo indica ao *nUnit* que o método definido a seguir deve ser executado antes de qualquer caso de teste. Já o *TearDown* é executado ao finalizar a execução de cada caso de teste. Ele pode ser utilizado, por exemplo, para liberar a memória ocupada pelo objeto instanciado. Nele é utilizado o atributo *[TearDown]* como exemplificado na linha 37, indicando ao *nUnit* que o método definido a seguir deve ser executado ao final de cada caso de teste.

Os demais métodos listados são os casos de teste. Para esses, é usado o atributo *[Test]* antes de sua definição, indicando ao *nUnit* que o método a seguir é um caso de teste.

No primeiro caso de teste, é definido um aluno que foi aprovado na prova final, tendo *nota1* igual a 3, *nota2* igual a 3, *notafinal* igual a 7 e *freqüência* em 75%, como pode ser visualizado a partir da linha 43. Já, na linha 47, é invocado o *IsTrue* da classe *Assert*. Essa classe é responsável por definir o valor esperado do teste para comparação com o valor obtido na sua execução. Neste caso, o valor esperado para retorno do método *calcularAprovacao* é verdadeiro (*IsTrue*).

O segundo caso de teste cobrirá a opção do aluno ser reprovado por freqüência. Para isso, é definida a variável *freqüência* para 74%. Os demais valores são definidos com 0, já que o aluno é reprovado de imediato. Para esse caso de teste, o valor esperado é falso, ou seja, *IsFalse*.

O terceiro caso de teste apresenta um aluno com 75% de freqüência, primeira nota com valor 3, segunda nota com 2.9 e prova final com 0, pois não importa o seu valor nessa situação. Para esse caso é esperado o valor falso (*IsFalse*).

Para o quarto caso de teste é configurado o aluno com 75% de freqüência, primeira e segunda notas com valor 3 e prova final com valor 6.9. Espera-se que o aluno seja reprovado, logo a classe *Assert* é configurada para invocar o *IsFalse*.

Por fim, o quinto e último caso de teste, o aluno é definido com 75% de freqüência e notas com valores iguais a 7 para a primeira e segunda nota. A nota final foi definida com 0. O valor esperado de retorno para o caso de teste é verdadeiro. Observe que a média é igual a 7.

A classe *Assert* apresenta outros métodos para definição de valores esperados além dos apresentados. Dentro os principais, são citados:

- *IsFalse*: se o parâmetro definido é falso;
- *IsTrue*: se o parâmetro definido é verdadeiro;
- *AreEqual*: se os parâmetros são iguais;
- *IsNotNull*: se o parâmetro não é nulo;
- *IsNull*: se o parâmetro é nulo.

Execução dos testes

Ao finalizar a programação dos casos de teste, deve-se complicar o projeto para que a DLL seja gerada. Após isso, a mesma deve ser carregada na interface do *nUnit* e *csUnit* para que ocorra a execução dos testes definidos. No *nUnit* o carregamento da DLL deve ser feito

Listagem 1. Unit Aluno

```
unit Aluno;

interface

type
  TAluno = class
  private
    frequencia, nota1, nota2, provafinal: Real;
  public
    procedure setDados (pFrequencia, pNota1, pNota2, pProvaFinal: real);
    function CalcularAprovacao: boolean;
    constructor Create;
  end;

implementation

function TAluno.calcularAprovacao: boolean;
var
  vMedia: real;
begin
  if self.frequencia < 75 then
  begin
    Result := False;
  end
  else
  begin
    vMedia := (self.nota1 + self.nota2) / 2;
    if vMedia < 3 then
    begin
      Result := False;
    end
    else
    begin
      if vMedia > 7 then
      begin
        Result := True;
      end
      else
      begin
        if (vMedia + self.provafinal) / 2 < 5 then
        begin
          Result := False;
        end
        else
        begin
          Result := True;
        end;
      end;
    end;
  end;
end;

constructor TAluno.Create;
begin
  inherited Create;
end;

procedure TAluno.setDados(pFrequencia, pNota1, pNota2, pProvaFinal: real);
begin
  self.frequencia := pFrequencia;
  self.nota1 := pNota1;
  self.nota2 := pNota2;
  self.provafinal := pProvaFinal;
end;

end.
```

acessando o menu *File>Open Project*. No *csUnit* o processo é semelhante: menu *Assembly>Add*. Após, solicite à ferramenta para que execute os testes.

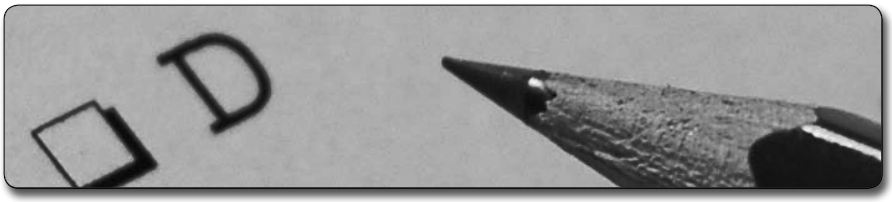
As **Figuras 6** e **7** apresentam, respectivamente, o resultado da execução dos testes nas ferramentas *nUnit* e *csUnit*. Observe que o último caso de teste falhou. Isso se deve ao fato que se esperaria que alunos com média 7 fossem aprovados e, na realidade, estão sendo reprovados. De acordo com o método *calcularAprovacao*, quem possuir média entre 3 e 7 (inclusive) deve fazer prova final. Como este aluno possui valor 0 na sua prova final, o mesmo foi reprovado, sendo retornado o valor falso pelo método, onde o valor esperado deveria ser verdadeiro, falhando assim o caso de teste.

Com isso foi possível identificar uma falha no código-fonte, uma vez que o aluno com média de valor 7 deveria ser aprovado. Deve-se então fazer uma alteração no código-fonte do método *calcularAprovacao* para que sejam aprovados os alunos com média superior ou IGUAL a sete. A alteração deve ser realizada na linha 34 da **Listagem 1**, modificando a condição para *vMedia >= 7*. A atribuição dos valores para os testes deve ser feita com muito cuidado. Por exemplo, se o caso de teste *TesteAprovacao5* utilizasse o valor 8 no lugar de 7 (linha 74 da **Listagem 2**), não seria possível encontrar o defeito no código-fonte da aplicação. Assim, sugere-se que sempre sejam utilizados os valores mais próximos das condições a serem testadas.

A **Figura 8** exibe a interface do *nUnit* com os resultados sem falhas após a alteração proposta. E a **Figura 9,** a execução dos testes no *csUnit*. Como foi observado, ambas as ferramentas têm a mesma funcionalidade e a sugestão é testar qual delas se adapta melhor para o projeto e ao profissional que a está utilizando.

Conclusão

Uma questão importante numa abordagem como essa é a definição do número de casos de teste necessários para avaliar os caminhos possíveis de um algoritmo. Como é impossível testar todas as situa-



Listagem 2. Unit AlunoTeste

```
01 unit AlunoTeste;
02
03 interface
04
05 uses
06   Aluno, NUnit.Framework;
07
08 type
09   [TestFixture]
10   TAlunoTeste = class
11   private
12     objAluno: TAluno;
13   public
14     constructor Create;
15     procedure SetUp;
16     procedure TearDown;
17     procedure TesteAprovacao1;
18     procedure TesteAprovacao2;
19     procedure TesteAprovacao3;
20     procedure TesteAprovacao4;
21     procedure TesteAprovacao5;
22   end;
23
24 implementation
25
26 constructor TAlunoTeste.Create;
27 begin
28   inherited Create;
29 end;
30
31 [SETUP]
32 procedure TAlunoTeste.SetUp;
33 begin
34   objAluno := TAluno.Create;
35 end;
36
38 [TEARDOWN]
39 procedure TAlunoTeste.TearDown;
40 begin
41   objAluno.Free;
42 end;
43
44 [TEST]
45 procedure TAlunoTeste.TesteAprovacao1;
46 begin
47   objAluno.setDados(75, 3, 3, 7);
48   Assert.IsTrue(objAluno.CalcularAprovacao,'Revise o Método de Aprovação');
49 end;
50
51 [TEST]
52 procedure TAlunoTeste.TesteAprovacao2;
53 begin
54   objAluno.setDados(74, 0, 0, 0);
55   Assert.IsFalse(objAluno.CalcularAprovacao,'Revise o Método de Aprovação');
56 end;
57
58 [TEST]
59 procedure TAlunoTeste.TesteAprovacao3;
60 begin
61   objAluno.setDados(75, 3, 2.9, 0);
62   Assert.IsFalse(objAluno.CalcularAprovacao,'Revise o Método de Aprovação');
63 end;
64
65 [TEST]
66 procedure TAlunoTeste.TesteAprovacao4;
67 begin
68   objAluno.setDados(75, 3, 3, 6.9);
69   Assert.IsFalse(objAluno.CalcularAprovacao,'Revise o Método de Aprovação');
70 end;
71
72 [TEST]
73 procedure TAlunoTeste.TesteAprovacao5;
74 begin
75   objAluno.setDados(75, 7, 7, 0);
76   Assert.IsTrue(objAluno.CalcularAprovacao,'Revise o Método de Aprovação');
77 end;
78
79 end.
```

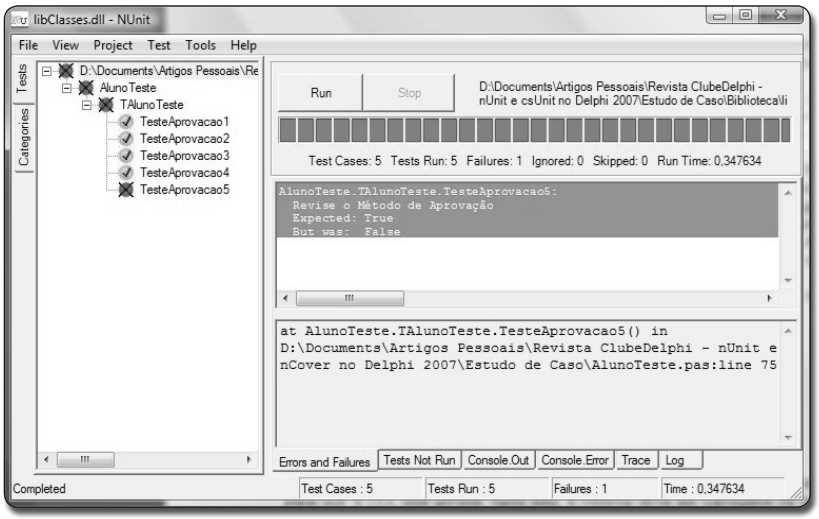


Figura 6. Resultados dos Testes no nUnit

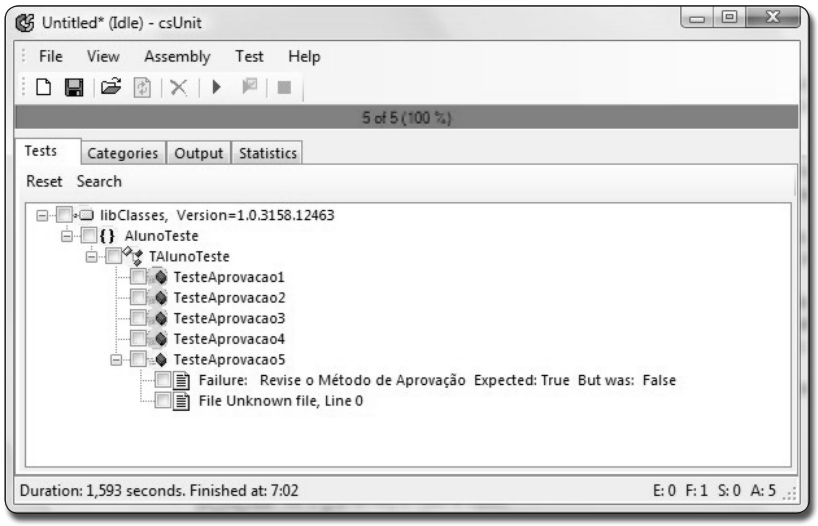


Figura 7. Resultados dos Testes no csUnit

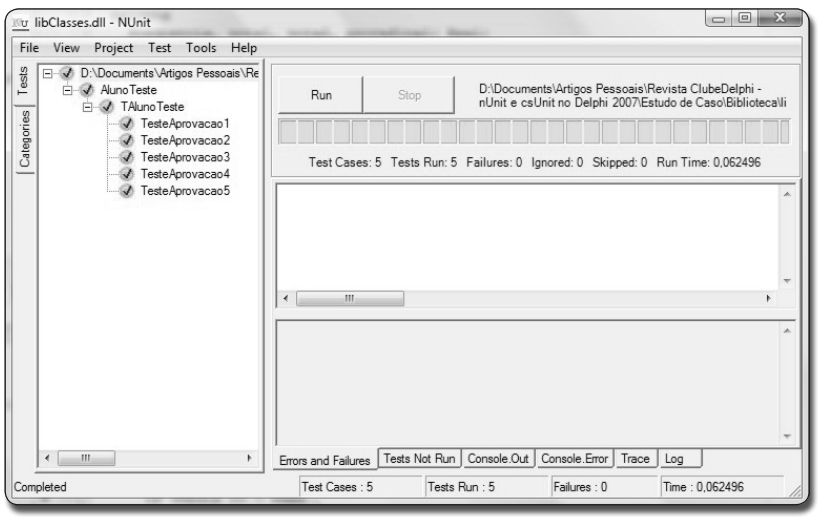


Figura 8. Resultados dos Testes no nUnit após correção do código

ções, torna-se primordial não construir casos de teste desnecessários, testando as mesmas situações, e nem deixar de construir um caso de teste para uma situação que necessite ser testada.

Uma importante técnica neste sentido é conhecida como Complexidade Ciclomática de McCabe, que avalia o número de caminhos possíveis em um algoritmo, indicando o número de casos de teste necessários para avaliar as suas possibilidades.

Entretanto, essa técnica apenas determina o número de casos de teste, não os valores que devem ser utilizados no mesmo. Para isso, uma outra técnica conhecida como Particionamento por Classes de Equivalência, auxiliada por outra técnica chamada Análise do Valor Limite, são úteis, mas fogem ao escopo deste artigo.

Entretanto, o sucesso de uma abordagem de desenvolvimento apoiada por testes dá-se em função da qualidade dos casos de teste. Se um sistema não falha nos testes planejados, fica a dúvida se o sistema é de boa qualidade ou se os casos de teste é que são de baixa qualidade. Portanto, investir num bom planejamento de testes é fundamental para esta estratégia.

O ambiente de execução de testes deve ser cuidadosamente planejado. A execução de testes em dias diferentes, por exemplo, pela simples mudança da data do sistema operacional, pode não representar a mesma situação a ser testada. Assim, é importante garantir que o ambiente seja o mesmo a cada execução do teste, preocupando-se, por exemplo, com o estado das informações no banco de dados.

Assim, a utilização de técnicas de teste proporciona aos desenvolvedores um menor índice de defeitos no código-fonte e, conseqüentemente, uma maior qualidade e confiabilidade do sistema no ambiente de produção.

Neste sentido, tanto o *nUnit*, quanto o *csUnit*, são *frameworks* para execução de testes unitários para o ambiente de desenvolvimento .NET, que buscam automatizar o processo de testes, tornando possível a implantação de uma política de testes na prática e sem agregar custos com ferramentas ao projeto, já que ambas são gratuitas. ●

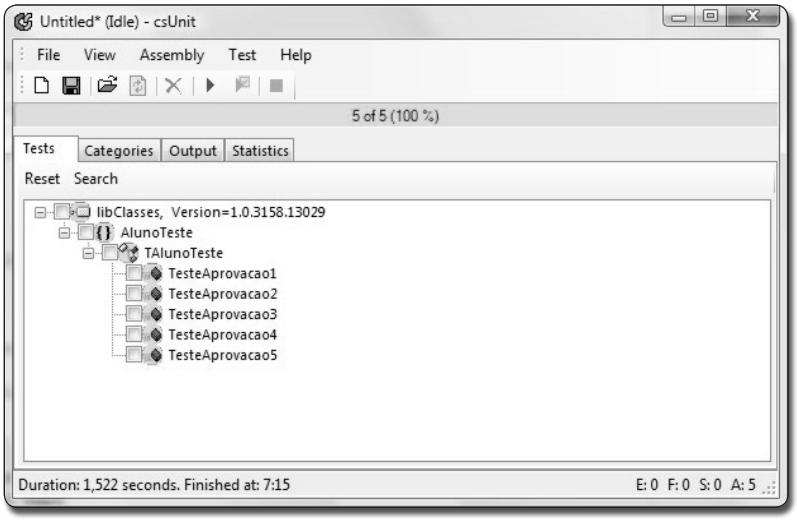


Figura 9. Resultados dos Testes no csUnit após correção do código

Links

Site Oficial do nUnit
www.nunit.org

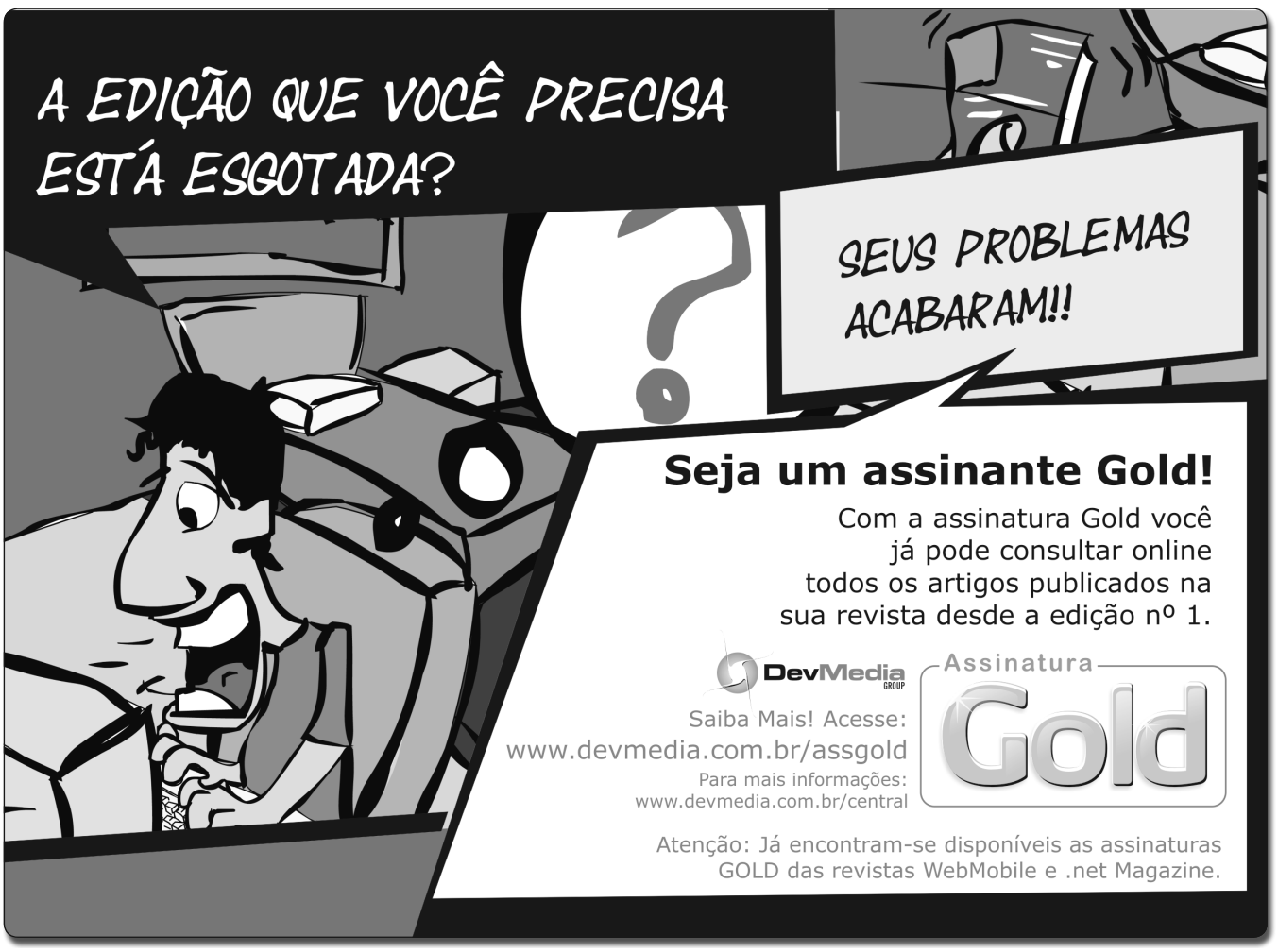
Site Oficial do csUnit
www.csunit.org

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback



Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software

Multicamadas

Ótimas razões para migrar sua solução client/server

Resumo DevMan

O desenvolvimento com o DataSnap no Delphi passou por várias fases. Entender como funciona o DataSnap internamente, suas vantagens e recursos, é essencial para a construção de aplicações corporativas, sejam elas multicamadas ou client/server. O DataSnap permite a construção de soluções mais robustas, com maior escalabilidade e performance, aliado aos poderosos recursos do COM+.

Nesse artigo veremos

- Arquitetura de aplicações multicamadas;
- Vantagens do desenvolvimento DataSnap e COM+;
- Funcionamento do dbExpress e DataSnap;
- Aplicações client/server com DataSnap local;
- Aplicações multicamadas com recursos avançados do COM+, como Object Pooling;

Qual a finalidade

- Mostrar como construir aplicações multicamadas baseadas em aplicações client/server.

Quais situações utilizam esses recursos?

- Mesmo aplicações client/server podem fazer bom uso de técnicas multicamadas. Uma aplicação dbExpress bem escrita é uma aplicação DataSnap local. Qualquer aplicação, seja ela planejada para uma solução corporativa ou não, pode fazer uso das técnicas aqui apresentadas.

Guinther Pauli

(guinther@devmedia.com.br)

é Delphi Certified: 6, 7, 2005, 2006, Web e Kylix e Microsoft Certfied: MCP, MCAD, MCSD.NET, MCTS e MCPD. Desenvolve aplicações multicamadas desde o MIDAS no Delphi 3/4. Já trabalhou em grandes projetos multicamadas, com mais de 3000 units clientes e 500 objetos servidores COM+.

O desenvolvimento multicamadas com o Delphi é algo encantador. Além de todas as facilidades que temos ao usar esse tipo de arquitetura, que cito a seguir, temos a facilidade do ambiente de desenvolvimento RAD do Delphi. Confesso que não vi até hoje um framework tão produtivo quanto o *DataSnap* para desenvolver soluções para tal propósito.

O objetivo deste artigo é apresentar técnicas intermediárias e avançadas de desenvolvimento com o *DataSnap* do Delphi, para que você definitivamente migre sua solução para multicamadas, mesmo que a solução vá rodar em uma única máquina. Apresentarei duas técnicas para este propósito, a de simular um ambiente *multi-tier* em *client/server* e como rodar o servidor como uma *library*.

Se o leitor lembrar, na edição 97, vimos como construir nossas primeiras aplicações multicamadas com o Delphi, fazendo chamadas remotas de procedimento usando COM+, sem, no entanto usar banco de dados. Neste artigo vou demonstrar técnicas de desenvolvimento *DataSnap*, ao mesmo tempo que apresento boas práticas de programação para essa arquitetura, usando agora um banco de dados. Partiremos do pressuposto que o leitor tenha uma aplicação *client/server* tradicional, com uma aplicação cliente *Win32* acessando um servidor *Firebird*.

Protocolos

Quando falamos em aplicações multicamadas, estamos falando em diferentes computadores se comunicando. Isso exige um protocolo. O primeiro passo é escolher qual o melhor protocolo para ser utilizado na comunicação. Atualmente, os protocolos suportados pelo *DataSnap* são:

DCOM – Você usa DCOM ao utilizar *Remote Data Modules*, ou *Data Modules* remotos. Não é uma boa solução pois não é escalável, visto que seus objetos são *state-full*, ou seja, para cada cliente há um objeto ativo no servidor, respondendo às requisições;

COM+ - Originado do *MTS*, o COM+ é a melhor solução para ambientes corporativos (intranet), pois é mais escalável, visto que seus objetos são *state-less*, ou seja, cada objeto é destruído no servidor após processar a requisição cliente;

SOAP – O suporte ao *Simple Object Access Protocol* surgiu no *Delphi 6*, e é recomendado quando sua solução precisa ter clientes espalhados pela Internet. É mais lento que o COM+ para ambientes corporativos, justamente por utilizar XML para intercâmbio de dados. Para mais informações sobre *DataSnap* com SOAP, recomendo a leitura do meu artigo na edição 70;

WebConnection – descontinuado pois desempenha basicamente a mesma função do SOAP;

CORBA – formato proprietário que teve a intenção de ser multi-plataforma. Descontinuado, recomendo a utilização do SOAP para estes casos;

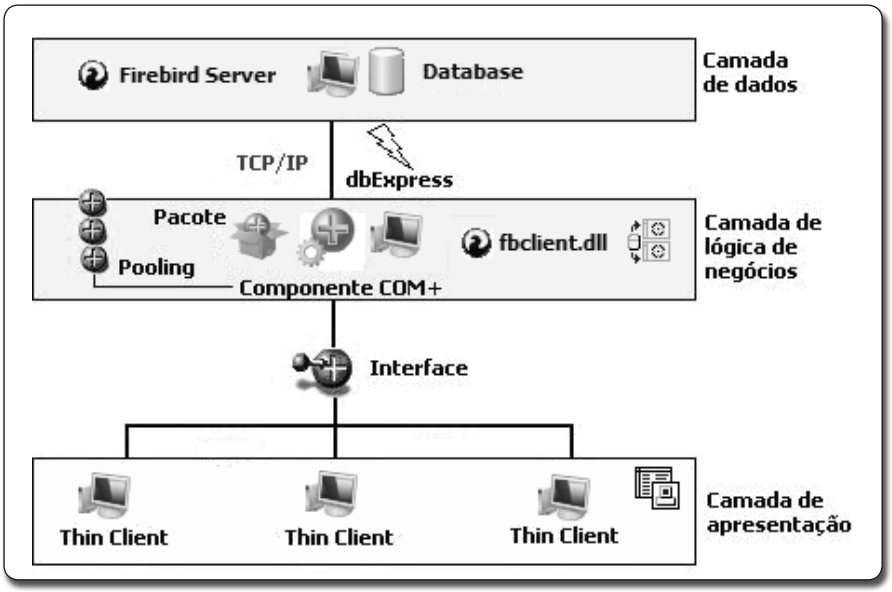


Figura 1. Solução multicamadas

BSS – Borland Socket Server, baseado no DCOM, também não recomendado para ambientes corporativos por ser *state-full*. Substituído pelo *TDsDataModule* no *Delphi 2009*;

Em nossa solução, vamos optar pelo melhor e mais óbvio, o COM+.

Arquitetura

Nada melhor que uma ilustração para representar como é formada uma solução *DataSnap*. Na **Figura 1,** temos um diagrama da solução que será construída. Ele servirá como nosso "mapa", sempre que tiver alguma dúvida sobre qual camada está sendo empregado determinado código, consulte a figura.

Em uma arquitetura multicamadas, temos vários computadores que desempenham papéis diferentes na solução. É óbvio que você pode usar um único computador para criar a aplicação *DataSnap*, porém as camadas físicas ainda existirão. Cada processo que roda em diferentes máquinas é também conhecido como *camada*. Em nossa figura, podemos identificar 3 camadas:

Camada de Dados – é o banco de dados em si, nesse caso, estamos utilizando o *Firebird*. Para acesso ao banco, o *engine* utilizado aqui é o *dbExpress* (nada impede que se utilize outro, como *ADO*, *IBX*, *Zeos*, *BDE* etc., desde que o *DataSet* suporte a interface *IProviderSupport*, discutida a seguir). Uma outra boa solução que recomendo, principalmente para novos projetos, é utilizar o *SQL Server* com *ADO*, pois se integra melhor ao COM+;

Camada Lógica de Negócios – também conhecido como *Servidor de Aplicação* (ou simplesmente *AppServer* para os mais íntimos), responsável por concentrar o acesso ao banco de dados e hospedar os componentes COM+. As aplicações cliente NÃO enxergam diretamente o BD, apenas se comunicam com o servidor de aplicação através de uma interface COM. É nessa camada que você centralizará as regras de negócio da aplicação, como por exemplo, validar uma compra, verificar a disponibilidade de um produto em estoque (consultando o BD), calcular algum imposto, realizar alguma validação etc. É nessa camada que residem os componentes do *dbExpress* e *DataSetProviders*;

Camada de Apresentação – é nessa camada que está a menor parte do código. Digo menor porque você basicamente criará telas para exibir os dados oriundos do servidor de aplicação e *database*. Essa camada não vê o BD, não tem *dbExpress* nem qualquer acesso direto ao servidor *Firebird*. Tudo é feito através da chamada de métodos COM ao servidor de aplicação. Por esse motivo, aplicações clientes em uma arquitetura *DataSnap* são também conhecidas como *Thin-Clients* (clientes leves ou magros);

Listagem 1. IAppServer

```
IAppServer = interface(IDispatch)
    ['{1AEFCC20-7A24-11D2-98B0-C69BEB4B5B6D}']
    function AS_ApplyUpdates(const ProviderName: WideString; Delta: OleVariant;

MaxErrors: Integer; out ErrorCount: Integer; var OwnerData: OleVariant): OleVariant; safecall;
    function AS_GetRecords(const ProviderName: WideString; Count: Integer;
      out RecsOut: Integer; Options: Integer; const CommandText: WideString;
      var Params: OleVariant; var OwnerData: OleVariant): OleVariant; safecall;
    function AS_DataRequest(const ProviderName: WideString; Data: OleVariant):
      OleVariant; safecall;
    function AS_GetProviderNames: OleVariant; safecall;
    function AS_GetParams(const ProviderName: WideString; var OwnerData: OleVariant):
      OleVariant; safecall;
    function AS_RowRequest(const ProviderName: WideString; Row: OleVariant;
      RequestType: Integer; var OwnerData: OleVariant): OleVariant; safecall;
    procedure AS_Execute(const ProviderName: WideString; const CommandText: WideString;
      var Params: OleVariant; var OwnerData: OleVariant); safecall;
end;
```

Como comentei, a comunicação entre as camadas exige um protocolo. Do servidor de aplicação para o BD, o protocolo utilizado é o TCP/IP, normal de uma comunicação *client/server* com o *Firebird*. O tipo de BD independe da solução, você pode usar *SQL Server, DB2, Oracle* etc.

A comunicação do *thin-client* com o servidor de aplicação é mais interessante. Ela usa um protocolo chamado COM (*Component Object Model*), através de uma interface. Imagine uma interface COM como um controle remoto de um DVD Player. Você apenas aciona comandos através dele, mas todo o processamento ocorre remotamente. O cliente precisa apenas conhecer a interface do servidor para se comunicar com ele, não toma conhecimento sobre como funcionam seus processos internos, como cálculos, validações e processamento de regras de validação, muito menos sobre o BD que está utilizando e como é feita a comunicação. No *DataSnap*, a interface base para a comunicação se chama *IAppServer*.

IAppServer

A interface *IAppServer* surgiu no Delphi 5, para substituir as então interfaces *IProvider* e *IDataBroker*. O motivo da substituição foi a inclusão do suporte a servidores *state-less*, como o caso do MTS/COM+. Além disso, houve uma falha de arquitetura ao se planejar o antigo *MIDAS* (versões 1 e 2). Para quem lembrar, cada *DataSetProvider* utilizado em um *Data Module* remoto exigia a criação de um novo método COM, o que obrigava a modificação da interface e a necessidade de um novo registro.

IAppServer resolve esse problema elegantemente, com uma solução simples: o nome do *DataSetProvider* é passado como parâmetro nas chamadas dos métodos COM, o que não exige um novo método para cada tabela que você queira acessar no banco. Além disso, a modificação de interfaces fere um dos princípios básicos da orientação a objetos, que diz que interfaces são imutáveis.

A interface *IAppServer* possui 7 métodos, e apenas 7, que fazem basicamente tudo o que você precisa em um servidor de aplicação, como solicitar dados, efetuar atualizações etc. A **Listagem 1** apresenta a referida interface.

Note que cada interface tem um GUID (*Global Unique Identifier - http://en.wikipedia.org/wiki/Globally_Unique_Identifier*), um identificador único de 128 bits que identifica a interface globalmente. Aqui o termo global significa mundo, é praticamente impossível existirem dois GUIDs iguais no globo.

Note também alguns importantes métodos na interface, como o *AS_GetProviderNames*. É esse método que é chamado, quando você baixa a lista *dropdown* da propriedade *ProviderName* de um *ClientDataSet* para ver os *DataSetProviders* disponíveis no servidor de aplicação ou locais. O *AS_GetParams* é acionado, quando você dá um clique de direita no *ClientDataSet* e chama o *FetchParams*, para solicitar que parâmetros de instruções SQL sejam passados na aplicação cliente (ou é acionado quando o método *FetchParams* do *ClientDataSet* é chamado em *runtime*).

AS_Execute é utilizado quando a propriedade *CommandText* do *ClientDataSet* é utilizada, ideal para passar instruções SQL diretamente da aplicação cliente. *AS_ApplyUpdates* é auto-explicativo, chamado quando atualizações são feitas no *ClientDataSet* e a seguir o método *ApplyUpdates* do mesmo componente é chamado. *AS_GetRecords* é invocado quando você chama por exemplo o método *Open* do *ClientDataSet*, para obter registros de uma tabela via servidor de aplicação ou localmente.

DataRequest é um método opcional para obter dados de um servidor de aplicação sem a necessidade da utilização de um *DataSetProvider* ligado a um *DataSet*, ideal no COM+ para não ser necessária a criação de métodos de negócio adicionais.

Note ainda que a interface *IAppServer* herda de *IDispatch*, interface que por sua vez herda de *IInterface* (a *IUnknown* renomeada). *IDispatch* é responsável por despachar chamadas remotas ao servidor de aplicação. Outro ponto notável é que todos os métodos estão com a convenção de chamada *safecall*, o que faz com que qualquer exceção levantada no servidor de aplicação seja empacotada e devolvida para o cliente. Caso contrário, se uma mensagem de erro fosse levantada no servidor e não no cliente, faria como que a solução inteira travasse, porque obviamente não existe um usuário operando o servidor de aplicação para pressionar o botão de Ok de uma exceção.

Como você pode ver, uma interface é apenas um mecanismo de definição, uma espécie de contrato que ambas as partes (cliente e servidor) devem conhecer e jogar suas regras. Ela não tem implementação (por isso mesmo é uma interface).

Quando formos criar nossos objetos COM+, uma nova interface será criada, descendente de *IAppServer*, onde poderemos incluir nossos métodos personalizados além dos 7 já pré-definidos. Quem implementa a interface é nosso objeto COM+, também conhecido como *CoClass*.

Vantagens do desenvolvimento multicamadas

Agora que conhecemos um pouco mais da arquitetura do *DataSnap* e *IAppServer*, vamos enumerar as principais vantagens do desenvolvimento multicamadas:

Centralização – todas as regras de negócio e acesso a banco de dados ficam

centralizadas no servidor de aplicação. Isso quer dizer que uma mudança em um requisito, que exija a mudança em algum código, não fará com que todos os clientes precisem ser recompilados e redistribuídos através da rede;

Modularização – justamente por sua natureza multicamadas, aplicações desse tipo são mais fáceis de manter e atualizar, pois funcionam como módulos;

Independência de BD – o cliente não enxerga o BD, aliás, nem sabe que tipo de BD está sendo utilizado em dado momento. Isso significa que você pode ter uma mesma solução adaptável a diferentes servidores de banco de dados, como *Firebird, SQL Server, MySQL, DB2, Oracle* etc. O BD pode inclusive ser um mero repositório de tabelas e relacionamentos, visto que a maioria do código para tratamento de dados pode ser escrito no servidor de aplicação, garantindo uma maior independência;

Performance – aplicações multicamadas foram feitas para serem rápidas, principalmente ao se utilizar o protocolo COM+;

Escalabilidade – aplicações multicamadas são mais escaláveis, ou seja, não perdem performance quando um número maior de usuários conecta a aplicação, muito disso tendo em vista a natureza *state-less* do COM+;

Economia de licenças – alguns bancos de dados exigem o pagamento de licenças extras para instalação de bibliotecas de conexão cliente. No *DataSnap* - que possui *royalts free* para distribuição de clientes desde o Delphi 7 -, as bibliotecas clientes ficam somente no servidor de aplicação, o que pode economizar licenças. Podemos ter 20 usuários simultâneos, por exemplo, usufruindo do mesmo cliente de banco (no caso do *Firebird* a biblioteca *fbclient.dll*). Soma-se a isso a facilidade de instalação do cliente, pois alguns bancos possuem clientes com centenas de MB, imagine instalar isso em cada estação cliente que acessará o servidor. No *DataSnap*, o cliente é só um EXE e pronto.

Vantagens do desenvolvimento COM+

Vamos agora enumerar algumas das vantagens ao se utilizar o COM+ como protocolo. As principais características do COM+ são gerenciamento, transações, escalabilidade, *pooling* e independência de cliente:

Componente	Name	Propriedades
SQLConnection	SQLConnection	LoginPrompt=False
SQLQuery	qryDepartment	SQLConnection=SQLConnection SQL= select * from DEPARTMENT Active=True
DataSource	dsDepartment	DataSet=qryDeparment
DataSetProvider	dspDepartment	DataSet=qryDepartment
ClientDataSet	cdsDepartment	ProviderName=dspDepartment
SQLQuery	qryEmployee	SQLConnection=SQLConnection SQL=select * from EMPLOYEE where DEPT_NO=:DEPT_NO DataSource=dsDepartment

Tabela 1. Componentes de acesso a dados do DataModule

Gerenciamento – O COM+ oferece um console para gerenciamento dos componentes, onde se pode gerenciar pacotes (coleções de componentes), instalar e registrar novos componentes, visualizar interfaces, métodos e instâncias ativas, e configurar modelos de transação. Desde o Delphi 6, para facilitar o gerenciamento de componentes COM+, há um componente chamado *COMAdminCatalog*, da paleta COM+. Com ele, sua aplicação pode se tornar um cliente para os componentes COM+ do sistema. Você pode, por exemplo, escrever um gerenciador próprio para o COM+ dentro de sua aplicação, ou instalar componentes automaticamente, sem precisar do console do COM+;

Transações – Com o COM+, você pode realizar transações em nível de objetos, em vez de usar apenas transações de banco de dados. Isso permite que métodos chamados em vários objetos diferentes sejam colocados no contexto de uma única transação, mesmo que os objetos não façam acesso direto a bancos de dados. Há suporte também a transações distribuídas, envolvendo vários bancos de dados. O mecanismo responsável por este gerenciamento de transações chama-se *MSDTC* (*Microsoft Distributed Transaction Coordinator*).

Escalabilidade – O COM+ permite que se aumente drasticamente o volume de aplicações concorrentes sem haver perda de desempenho na camada de negócio, tudo graças ao mecanismo de *pooling* de objetos e ao modelo *state-less*;

Pooling – O COM+ pode manter 5 instâncias de um objeto atendendo a mil clientes, por exemplo. É isto que o torna tão escalável. Isso é feito através do mecanismo *state-less* e do *pooling* de objetos. Quando um cliente faz uso de um objeto, é o COM+ que gerencia como sua instância se comportará na memória. Com isso, ele pode liberar um objeto sem o conhecimento do cliente. Mas, e quando o cliente chamar um método de um objeto e ele não existir? O COM+ instancia-o novamente ou usa um objeto que já esteja no *pool*. Com o mecanismo *state-less*, objetos não mantêm o estado: eles não "se lembram" das últimas operações realizadas pelo cliente. Isso torna possível otimizar a instanciação dos objetos e o acesso a eles;

Independência de cliente – Outro aspecto a se observar é o fato de você poder utilizar um objeto COM+ feito em Delphi a partir de outras linguagens, como VB, ASP, C++, .NET etc. Com isso, você pode construir um poderoso *middleware* em *Delphi* para distribuir objetos a serem chamados em código ASP, por exemplo. Aliás, é muito comum desenvolvedores *Delphi* construírem componentes COM usando recursos da VCL, para serem utilizados em páginas ASP ou ASP.NET.

Uma aplicação Client/Server

Agora que conhecemos um pouco mais sobre o *DataSnap* e COM+, vamos construir um exemplo prático. Aqui utilizarei o *Delphi 7*, mas fique à vontade para utilizar outra versão, como o *Delphi 2007/2009*. Inicie uma nova aplicação *VCL Win32* e salve todos os arquivos em uma pasta chamada *Client*. Através do menu *File>New* crie um novo *DataModu-*

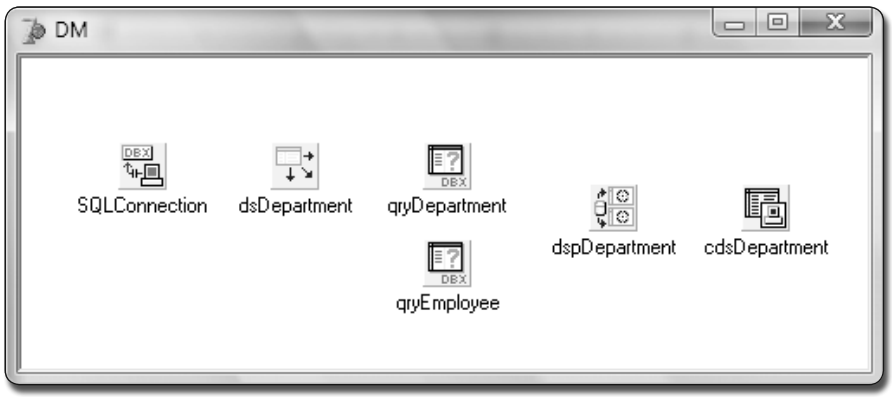


Figura 2. DataModule da aplicação

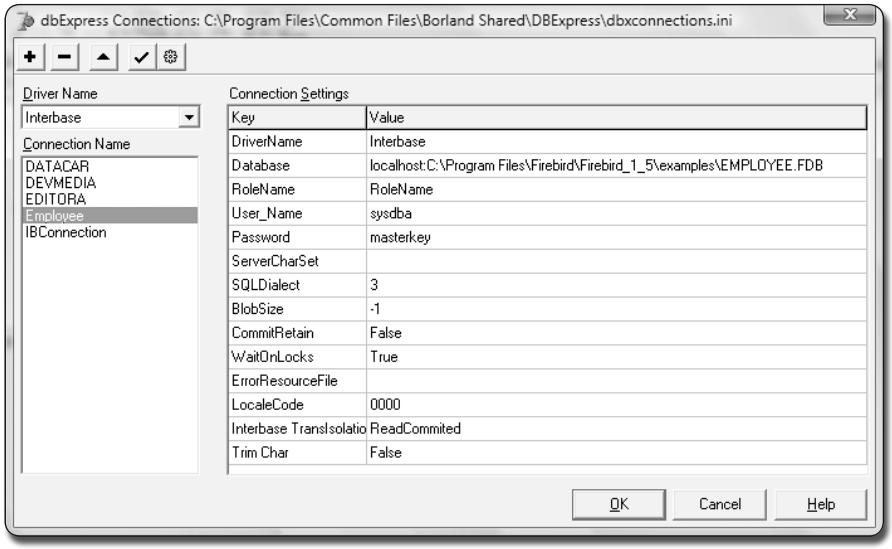


Figura 3. Configurando a conexão

le, dando a ele o nome de *DM*. Usando as paletas *dbExpress* e *Data Access* coloque e configure os componentes descritos na **Tabela 1.** Ao final seu *DM* deve estar semelhante ao da **Figura 2**.

Dê um duplo clique no *SQLConnection*, no editor clique no botão +, selecione o driver *Interbase* e em *Name* digite *Employee*. Configure os parâmetros de conexão como mostrado na **Figura 3.** Note a propriedade *Database* que aponta para o banco *Employee.fdb* que acompanha o *Firebird*. Clique em Ok. Na propriedade *VendorLib* do *SQLConnection* digite *fbClient.dll*, que é a biblioteca cliente nativa do *Firebird*. Certifique-se que essa DLL esteja no *path* do *Windows*, ela pode ser encontrada no diretório de instalação do *Firebird*.

Vá na propriedade Params do qryEmployee e configure o parâmetro DEPT_NO como sendo ParamType=ptInput e DataType=ftInteger.

Vamos entender o que foi feito até aqui. Os componentes colocados configuram uma relação *master/detail* entre as tabelas *Department* (departamentos) e *Employee* (empregados) do banco. Ou seja, um departamento possui vários empregados. O interessante dessa relação é que usamos um *DataSource* apontando para a *query* de departamentos, e a propriedade *DataSource* (que pode parecer estranha estar presente em um *DataSet*) aponta para o *dsDepartment*. Com isso, o *dbExpress* entende que há uma relação mestre/detalhe entre as duas tabelas, e o *DataSetProvider* pode empacotar o conteúdo de ambas as tabelas em um único *DataPacket*.

Dê um duplo clique em *qryDepartment* e no editor aperte *Ctrl+A* e confirme em

Nota do DevMan

DataPackets são pacotes de dados que trafegam através das chamadas de procedimento remoto do *DataSnap*. Por exemplo, os dados de uma tabela, no servidor de aplicação, são empacotados em um formato específico (binário ou XML) para serem enviados pela rede ao cliente. Esse pacote é também conhecido como *Data*. No *Delphi*, quem cuida de criar o *DataPacket* é a biblioteca MIDAS.DLL. O segredo do empacotamento, por ser criar pacotes pequenos e otimizados, é um segredo que a equipe do *Delphi* guarda até hoje. Os pacotes também podem estar em formato XML, no caso de aplicações SOAP. Para trabalhar com intercâmbio de dados XML do *DataSnap* com plataformas diferentes, usamos o utilitário *XML Mapper* (veja edições 36 e 100). Quando alterações são feitas no *ClientDataSet* e repassadas de volta ao servidor de aplicação, elas são novamente empacotadas, num *DataPacket* conhecido como *Delta*.

Nota do DevMan

TFields são objetos que representam campos de um *DataSet*. É possível utilizá-los de duas formas, ou acessando o método *FieldByName* ou a collection *Fields* do *DataSet*, para obter um objeto *TField* (classe base de todos os *TFields* do Delphi) ou ainda adicionando os campos fortemente tipados (*TIntegerField*, *TStringField*) no próprio *DataSet*, dando um duplo clique sobre ele.

Nota do DevMan

O que significa o parâmetro "0" (zero) passado no *ApplyUpdates*? Quando fazemos atualizações no *ClientDataSet*, nada é aplicado no servidor até você chamar *ApplyUpdates*. Tudo fica em uma cache local. Normalmente um registro é atualizado no BD logo após o POST, mas isso não é obrigatório. O zero do *Apply* significa que a operação deve falhar caso algum erro seja percebido na atualização de pelo menos um registro da *cache delta*. Para aplicar a maior quantidade de registros possíveis, passamos -1. Os erros originados desse *Apply* devem ser tratados no evento *OnReconcileError* do *ClientDataSet*.

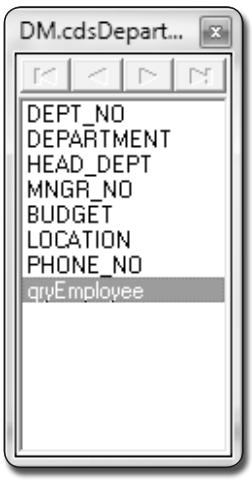


Figura 4. DataSetField

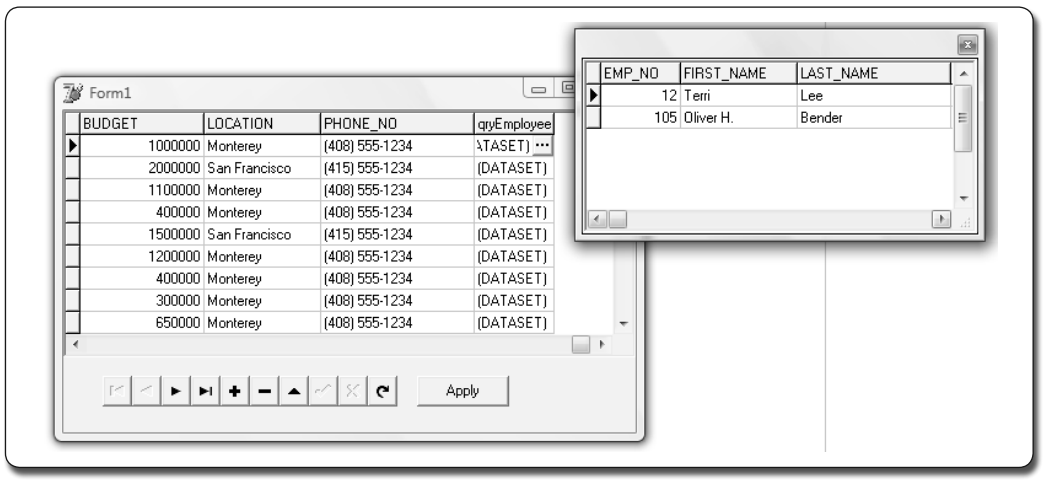


Figura 5. Aplicação em funcionamento

Ok para adicionar os campos *TFields* ao *DataSet*. Repita o mesmo procedimento para o *qryEmployee* e *cdsDepartment*.

Ao adicionar os *TFields* no *ClientDataSet*, você deve ter notado que além de todos os campos da tabela *Department* (**Figura 4**), há também um campo com o nome da *query detail* da relação. Esse é um campo que representa nada mais nada menos que a tabela inteira de empregados. Esse recurso é conhecido no Delphi como *DataSetFields*. Além disso, o *DataSnap* é capaz de empacotar cada departamento juntamente com seus respectivos empregados (daí a necessidade de um único *DataSetProvider* para ambas as tabelas).

Criando a interface

No formulário principal da aplicação coloque um *DataSource* (*Data Access)* e um *DBGrid (Data Controls)*. Aponte o *DBGrid* para o *DataSource* através da propriedade de mesmo nome. Clique em *File>Use Unit* e selecione a unit do *DM* (que deve ser a *unit2*). Aponte o *DataSource* do *form* para o *cdsDepartment*. Coloque também um *DBNavigator (DataControls)* e aponte-o para o *DataSource* do *form*. Coloque um *Button* e no seu evento *OnClick* digite:

```
procedure TForm1.ApplyClick(Sender: TObject);
begin
  DM.cdsDepartment.ApplyUpdates(0);
end;
```

Dê um duplo clique no *form* e no seu evento *OnCreate* digite:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DM.cdsDepartment.Open();
end;
```

Antes de executar a aplicação, clique em *Project>Options* e certifique-se que o *DM* seja criado antes do *form*. Pressione *F9* para executar a aplicação (**Figura 5**).

Se quiser ocultar a palavra *(DATASET)* que aparece como conteúdo do último campo, basta selecionar o campo *DataSetField* no editor de campos do *ClientDataSet* e no seu evento *OnGetText* digitar:

```
procedure TDM.cdsDepartmentqryEmployeeGetTex
t(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
   if Sender = cdsDepartmentqryEmployee then
     Text := '';
end;
```

Como funciona o dbExpress e DataSnap

Vejamos agora como funciona do *DataSnap* / *dbExpress*. Mesmo que você esteja utilizando uma aplicação *client/server* normal, o funcionamento a seguir explicado ainda vale. Isso porque o *dbExpress* local funciona de forma muito semelhante ao *DataSnap* multicamadas, por fazer uso dos componentes *ClientDataSet* e *DataSetProvider*. A **Figura 6** demonstra esse funcionamento. Considerando uma operação simples de *select* e atualização, temos os seguintes passos executados:

1 – O método *Open* do *ClientDataSet* é chamado. Isso faz internamente uma chamada a *AS_GetRecords* ser invocada. Nessa chamada a *IAppServer*, o nome do *provider* indicado na propriedade *ProviderName* é passado ao servidor, que se encarrega de localizar o componente *DataSetProvider* associado;

2 – No servidor de aplicação, o *DataSetProvider* dispara importantes eventos, como *BeforeGetRecords*, caso você queira fazer alguma operação antes dos dados serem obtidos a partir do *engine* de conexão;

3 – O *DataSetProvider* se comunica com o *engine* de conexão, nesse caso o *dbExpress*, para obter os dados o *select*. Ele vai abrir um cursor unidirecional de dados indicado pela sua propriedade *DataSet*. Essa comunicação com o *DataSet* associado é feito também através de uma interface, chamada *IProviderSupport*. Como podemos ver, todos os *DataSets* do Delphi devem implementar essa interface para dar suporte ao empacotamento e reconciliação de dados. Comprovamos isso pela declaração da classe *TDataSet* na unit *DB.pas* da *VCL*:

```
TDataSet = class(TComponent, IProviderSupport)
```

A interface *IProviderSupport* é mostrada na **Listagem 2**. Nela podemos observar importantes métodos como *PSExecute* e *PSExecuteStatement*, encarregado de executar comandos SQL, *PSGetParams*, para recuperar uma lista de parâmetros para ser enviada do *DataSet* servidor ao *ClientDataSet* local, *PSUpdateRecord* para atualizar um *DataPacket* de dados etc. Esses métodos são todos declarados como virtuais em *TDataSet*, o que significa que o *DataSet* descendente, como o *TSQLDataSet*, deve sobrescrevê-los;

4 – O *DataSetProvider*, via interface *IProviderSupport*, chama o método *Open* do *DataSet* que pode estar fechado, que por sua vez abre a conexão com o BD

Listagem 2. Interface IProviderSupport

```
IProviderSupport = interface
['{7AFt8F684-0660-47B5-A1B3-E168D2ACB908}']
  procedure PSEndTransaction(Commit: Boolean);
  procedure PSExecute;
  function PSExecuteStatement(const ASQL: string; AParams: TParams;
    ResultSet: Pointer = nil): Integer;
  procedure PSGetAttributes(List: TList);
  function PSGetDefaultOrder: TIndexDef;
  function PSGetKeyFields: string;
  function PSGetParams: TParams;
  function PSGetQuoteChar: string;
  function PSGetTableName: string;
  function PSGetIndexDefs(IndexTypes: TIndexOptions = [ixPrimary..ixNonMaintained]): TIndexDefs;
  function PSGetUpdateException(E: Exception; Prev: EUpdateError): EUpdateError;
  function PSInTransaction: Boolean;
  function PSIsSQLBased: Boolean;
  function PSIsSQLSupported: Boolean;
  procedure PSReset;
  procedure PSSetParams(AParams: TParams);
  procedure PSSetCommandText(const CommandText: string);
  procedure PSStartTransaction;
  function PSUpdateRecord(UpdateKind: TUpdateKind; Delta: TDataSet): Boolean;
```

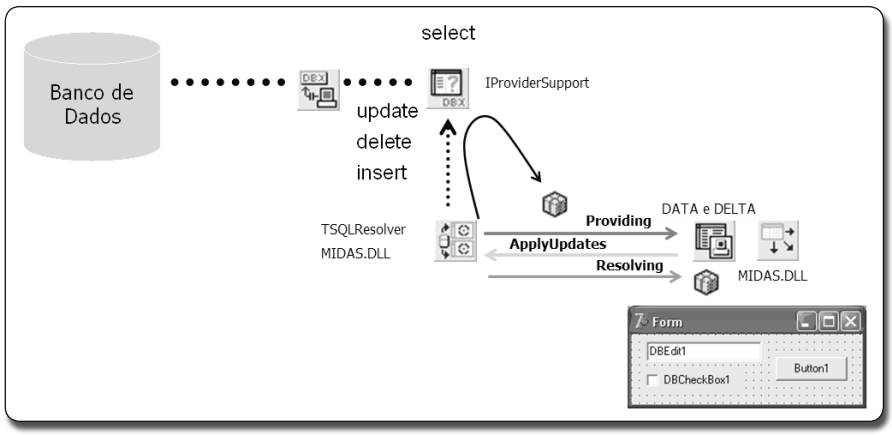


Figura 6. Funcionamento do dbExpress e DataSnap

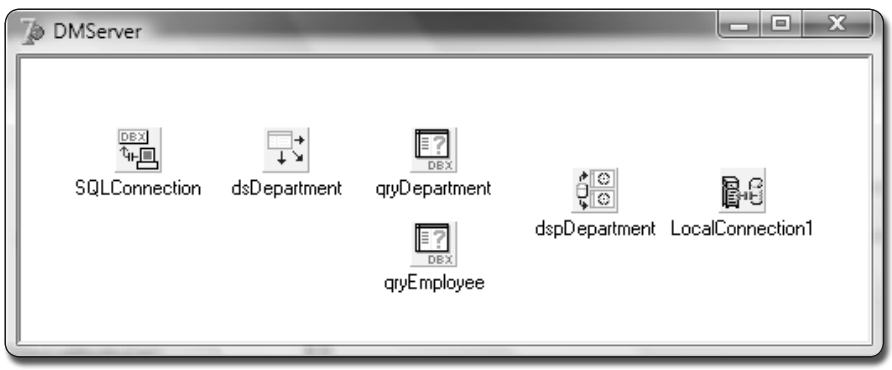
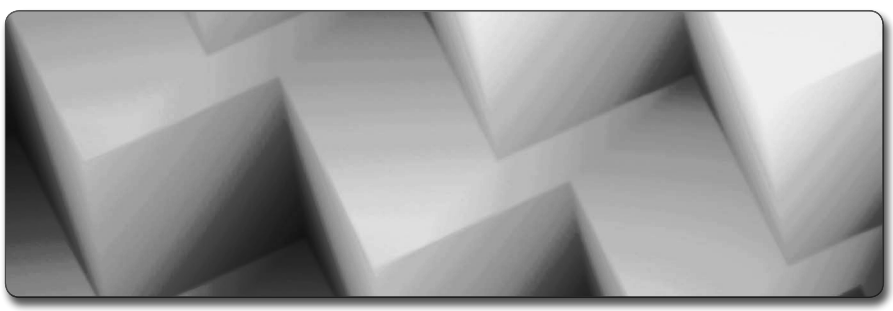


Figura 7. DMServer simula um AppServer local



via *SqlConnection*. Um cursor de dados é aberto no servidor, e o *DataSetProvider* inicia a varredura desse cursor (na verdade é um *while not eof* no *DataSet*). Ele vai então criando um pacote de dados (*DataPacket*) chamado *Data*. Internamente, quem cria esse pacote *Data* é a classe *TCustomPacketWriter* da unit *Provider.pas*, auxiliada pela biblioteca MIDAS.DLL. A comunicação com a MIDAS.DLL também é feita através de interfaces, como de costume, como pode ser visto na unit *IDSWriter.pas* da *VCL;*

5 – Criado o *DataPacket*, o *Data* é enviado através do *DataSnap* para o respectivo *ClientDataSet*, que se encarrega de guardar em sua propriedade *Data* os dados oriundos do *AppServer*. Os dados são exibidos ao usuário;

> **Nota**: O processo de comunicação, troca de dados e serialização de dados entre diferentes processos é conhecido como *marshalling*.

6 – O usuário localmente faz várias atualizações em vários registros do *DataPacket*, que vão ficando guardados em memória, em um outro pacote chamado *Delta*. Quando o método *ApplyUpdates* é chamado, há uma comunicação com *IAppServer* através do método *AS_ApplyUpdates*. O *Delta* é passado então ao

Nota do DevMan

Como o *TSQLResolver* sabe montar as instruções SQL de atualização? Para isso o *DataSetProvider*, através da interface *IProviderSupport*, consulta a propriedade *ProviderFlags* de cada campo *TField* do *DataSet* indicado na propriedade de mesmo nome, normalmente um *SQLDataSet* ou *SQLQuery*. Nesse *TField*, o campo indica nas opções *pfInWhere, pfInKey e pfInUpdate* se deve ser colocado nas cláusulas *update/insert/delete e where* de cada comando de atualização gerado. Normalmente, o campo chave tem *pfInKey* e *pfInWhere* igual a *True*. Os demais campos têm o *pfInWhere* como *False*. Campos oriundos de *joins*, como descrições, normalmente não são atualizados (somente a *Foreign-key*), então têm o *pfInUpdate* como *False*.

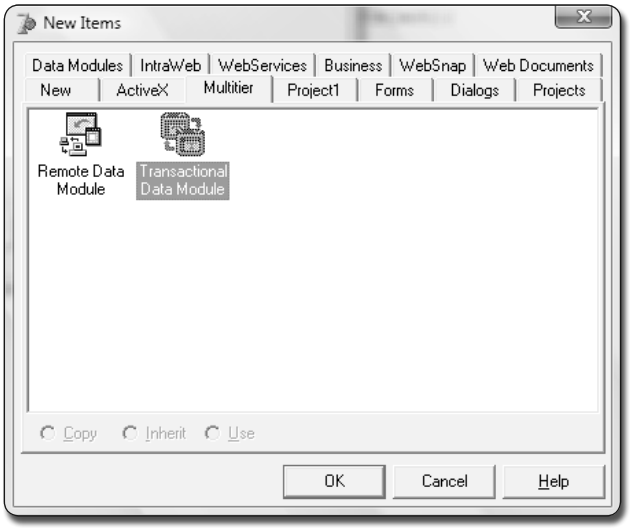


Figura 8. Criando o servidor de aplicação

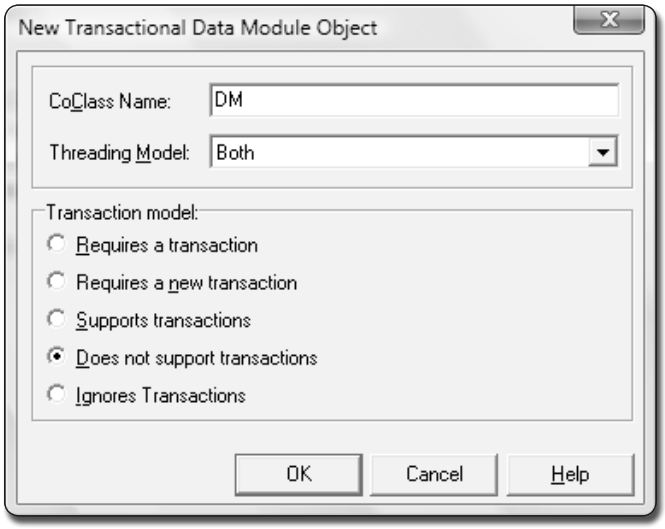


Figura 9. Parâmetros do servidor

servidor de aplicação e recebido pelo *DataSetProvider*;

7 – O *DataSetProvider* inicia então um processo chamado *Resolving*, que consiste em desmembrar os registros do *Delta* em instruções SQL de *insert, update* e *delete*. Antes de cada operação, ele chama o importante evento *BeforeUpdateRecord*, indicando no parâmetro *UpdateKind* o tipo de atualização que está sendo feita. É nesse ponto que podemos efetuar uma validação por exemplo, ou resolver uma atualização com *join* onde mais de uma tabela sofre um *update/insert/delete*. Novamente, essa comunicação é feita com o *DataSet* do *dbExpress* através da interface *IProviderSupport*. A geração dos comandos SQL de atualização é feita automaticamente. Salvo várias exceções, no *dbExpress* você raramente escreve *updates*, *inserts* e *updates*, apenas o *Select*. Isso se deve graças a classe *TSQLResolver* da unit *provider.pas*, que monta as instruções SQL no momento do *Resolving*. Outros importantes eventos, como *BeforeApplyUpdates*, são chamados no *DataSetProvider*;

8 – O *SqlDataSet* tenta então executar as instruções SQL passadas via *IProviderSupport*. Se houver algum erro gerado no SGBDR, como a violação de uma *ForeignKey*, um novo processo é desencadeado, chamado *Resolving*. Os registros problemáticos são devolvidos ao *ClientDataSet* para serem corrigidos, procedimento feito no seu evento *OnReconcileError*.

DataSnap Local

Podemos começar nossa migração de *client/server* para *DataSnap* usando um *AppServer* local. Crie um novo *DataModule* chamado *DMServer*. Agora dê um *Ctrl+X* nos componentes *dbExpress* que estão no DM (*SQLQuerys* e *SQLConnection*) e também *DataSource* e *DataSetProvider*, colando-os no *DMServer*. Da paleta *DataSnap* coloque um componente *LocalConnection*. Seu *DMServer* deve estar semelhante ao da **Figura 7**.

Nota do DevMan

Por que a propriedade *ProviderName* do *ClientDataSet* é uma string e não uma referência para um objeto, como a propriedade *DataSet* de um *DataSource*? Uma referência a objeto não faz sentido em uma aplicação multicamadas, pois um objeto estará tentando referenciar um endereço de memória inválido rodando em outro processo em outra máquina. Por esse motivo *ProviderName* é uma string, que é passada pelo método *AS_GetRecords* de *IAppServer* quando o método *Open* do *ClientDataSet* é chamado. Por esse motivo, se você alterar o nome de um *DataSetProvider* depois de ter configurado o *ProviderName*, deverá manualmente refazer a referência, pois o IDE não toma conhecimento da mudança como o que acontece com ponteiros diretos para objetos.

Volte ao *DM* e clique em *File>Use Unit*, escolhendo a *unit* do *DMServer* (que deve ser a *unit3*). Agora selecione o *ClientDataSet* e em *RemoteServer* escolha *DMServer.LocalConnection1*e mantenha o valor de *ProviderName*. Clique em *Project>Options* e certifique-se que a ordem de criação dos itens seja *DMServer, DM* e *Form1*. Execute e teste a aplicação.

Nota: Uma opção ao uso do *LocalConnection* é chamar o método *SetProvider* do *ClientDataSet* e passar o *DataSetProvider* como parâmetro, assim:

cdsDepartment.SetProvider(DMServer.dspDepartment)

Configurar a propriedade *ProviderName* manualmente não funciona.

DataSnap em ação: criando o servidor

Vamos iniciar o desenvolvimento multicamadas propriamente dito. Clique em

Nota do DevMan

O que é uma CoClass? CoClass é uma classe COM+ que implementa uma interface COM. Por exemplo, um TransactionalDataModule ou um RemoteDataModule são ambos CoClasses.

File>New>Other e na aba *Multitier* escolha *Transactional Data Module* (**Figura 8**). Na mensagem que aparece, informando que será criado um projeto *ActiveX*, clique em Ok. Na próxima tela (**Figura 9**), dê o nome de *DM* para a *CoClass* e escolha o tipo *Both* para *Threading Model*. Isso é necessário para que possamos usar o recurso de *Object Pooling* visto a seguir. Clique em Ok e salve todos os arquivos em uma pasta chamada *Server*. Veja que foi criado um *DataModule*.

Agora abra apenas o *DMServer* da pasta *Client*, dê um *Ctrl+X* nos componentes que cole no *DataModule* da aplicação *DataSnap*. Retire o *LocalConnection* porque ele não será mais necessário aqui.

Uma boa dica ao se trabalhar com aplicações *DataSnap* é criar um grupo de projetos (*Project Group*). Para isso, vá até a opção *View>Project Manager*, dê um clique de direita sobre o grupo de projetos e escolha *Add Existing Project*, escolhendo o projeto cliente. Salve o grupo com a extensão BPG. Agora ambos os arquivos dos projetos po-

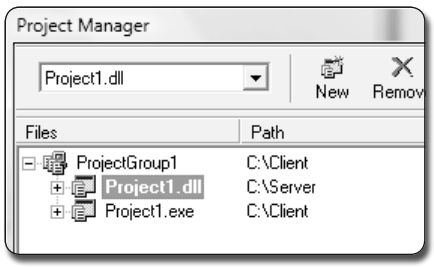


Figura 10. Grupo de projetos

dem ser facilmente acessados, (**Figura 10**), não é necessário fechar a aplicação servidora para trabalhar com um formulário da aplicação cliente, por exemplo. Com o servidor selecionado, clique em *Project>Build* para compilar a DLL do servidor COM+.

Registrando o servidor

Agora precisamos registrar o servidor de aplicação para que possa ser acessado por aplicações clientes. No *Painel de Controle>Ferramentas Administrativas*, acesse os *Serviços de Componentes*. Usuários do Windows Vista devem digitar no *Executar*:

```
C:\windows\system32\comexp.msc
```

O que você está vendo é o gerenciador do catálogo do COM+. Abra a árvore até encontrar o item referente ao seu computador, clique de direita e *COM+ Applications* e escolha *New>Application*. No assistente clique em *Next*, depois clique em *Create an empty application* (**Figura 11**), dê um nome para a aplicação (*ClubeDelphi*) e finalize o assistente.

De volta ao gerenciador, vá até a aplicação criada e clique de direita em *Components*, escolhendo a seguir *New>Component*. No gerenciador que abre clique em *Next* e depois *Install new component(s)* (**Figura 12**). Escolha então a TLB (*Type Library*, discutida a seguir) do projeto servidor e finalize o assistente. Seu componente deve estar registrado conforme a **Figura 13**.

> **Dica**: você pode usar esse gerenciador para administrar objetos de outras máquinas, bastando para isso registrar o computador remoto.

Testando o cliente

Agora na aplicação cliente remova o *DMServer*, pois não usaremos mais um *DataModule* local, mas sim o *Transactional Data Module* da aplicação servidora. Para isso acesse o *Project Manager*, clique de direita sobre a *unit* desse *DataModule* (que deve ser a *unit3*) e escolha *Remove from Project*. Agora, do *DM* remova o *uses unit3* da cláusula *uses* (sugiro também apagar o *DMServer* do disco para não haver dúvidas que vamos usar o *DataModule* remoto e não o local).

Agora, no *DataModule* da aplicação cliente, onde deve ter sobrado apenas um *ClientDataSet*, coloque um *DCOMConnection* da guia *DataSnap*. Em *ServerName* escolha o nosso *Transactional DataModule*, que deve ser algo como *Project1.DM*. Note que a propriedade *ServerGuid* já é configurada automaticamente. Aponte o *RemoteServer* do *ClientDataSet* para o *DCOMConnection* e execute a aplicação. Note que tudo funciona perfeitamente, só que agora usando 3 camadas.

Nesse momento o objeto será levantado no servidor. A nossa DLL do *Transactional DataModule* ficará hospedada em um processo servidor chamado *DLLHost.exe*. Você pode ver esse processo em

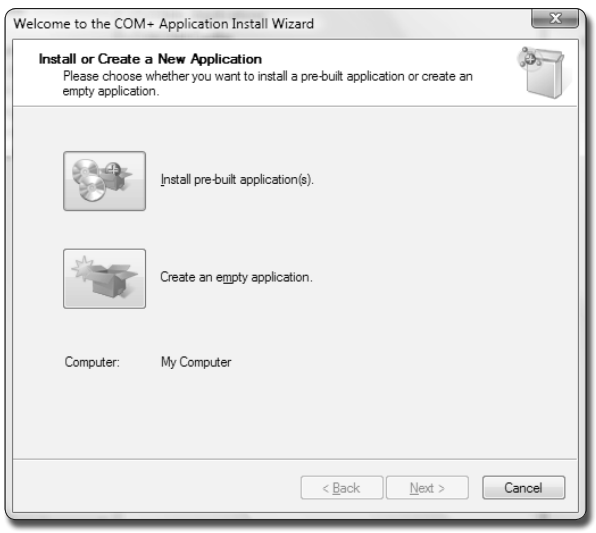


Figura 11. Criando uma aplicação COM+

Figura 12. Instalando componentes

execução pelo *Task Manager* do Windows (ou ainda pelo utilitário bem melhor chamado *Process Explorer*). Se você tentar recompilar a aplicação servidora, receberá um erro dizendo que o arquivo DLL não pode ser criado:

```
[Fatal Error]
Could not create output file 'Project1.dll'
```

Isso acontece porque a DLL está em execução, hospedada no *DLLHost* (**Figura 14**). Para tirá-la da memória, você precisa no catálogo do COM+ dar um clique de direita na aplicação (*ClubeDelphi*) e escolher *Shutdown* (*Desligar*). Note que o ícone que representa o objeto COM+ fica animado enquanto estiver em execução.

> **Nota**: Um bom atrevimento é finalizar o processo *DLLHost.exe* manualmente para recompilar o servidor.

Object Pooling

Um recurso interessante do COM+ chama-se *Object Pooling*. Esse recurso permite que objetos sejam mantidos em memória no servidor após serem executados, e são reutilizados no momento de sua ativação. Como o COM+ é *state-less*, em algumas situações é vantajosos ativar o *Pooling* para que usuários, após terem utilizado objetos, deixem-os prontos e criados para serem utilizados por outros usuários.

Imagine uma conexão com banco de dados. Se para cada usuário que conectar no servidor de aplicação abríssemos uma conexão, isso poderia degradar a performance. A abertura da conexão com o BD é um dos recursos mais caros da programação. Então porque não guardar a conexão em *pool* (uma espécie de *cache*) e reutilizá-las para múltiplos usuários? Todo o tempo necessário para estabelecer uma conexão TCP/IP com o servidor de banco de dados e autenticar o usuário pode ser eliminado, visto que quando um usuário solicitar dados, já pegará uma conexão pronta do *pool*.

Vejamos como isso funciona na prática. No catálogo do COM+ dê um clique de direita no nome da aplicação e na aba *Pooling & Recycling* coloque o valor *10*

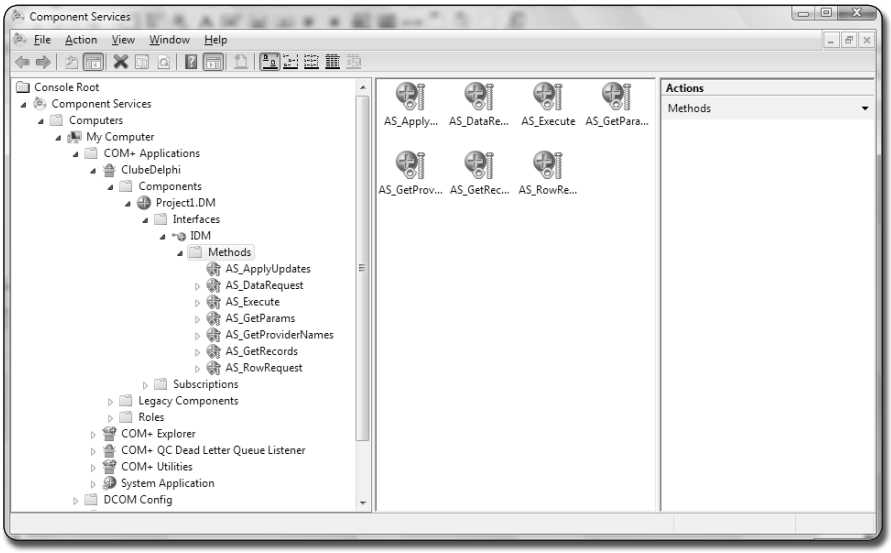


Figura 13. Componente instalado

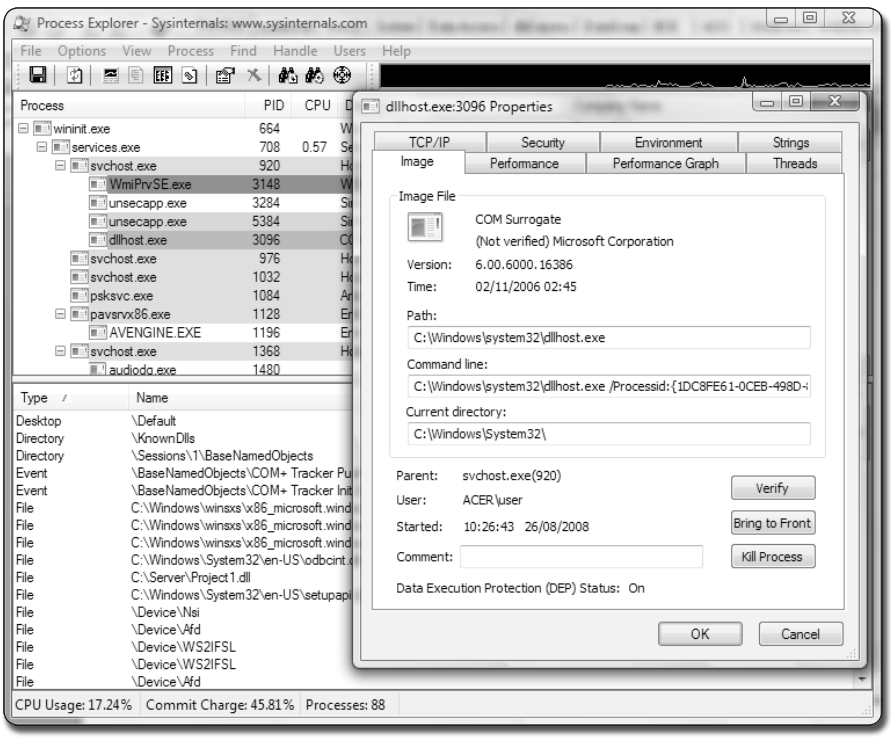
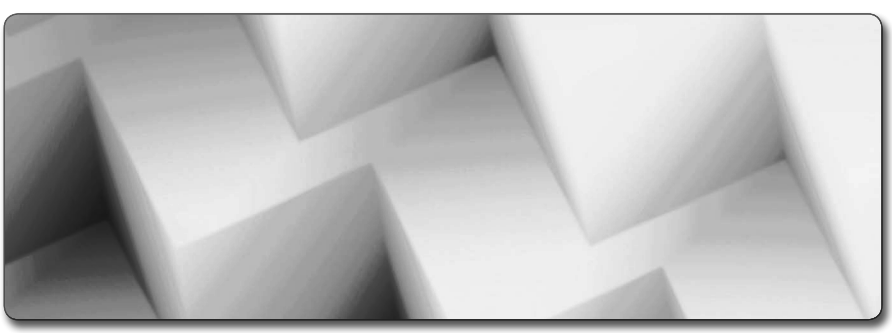


Figura 14. Processo DLLHost.exe hospeda nosso servidor COM+



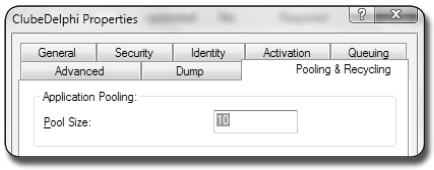


Figura 15. Ativando o Object Pooling na aplicação

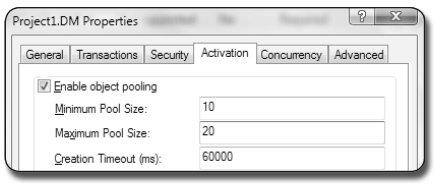


Figura 16. Ativando o Object Pooling no componente

(**Figura 15).** Esse passo é necessário para configurar o *Object Pooling* na aplicação.

O próximo passo é configurar o *Object Pooling* no componente. Para isso dê um clique de direita no componente (*Project1.DM*) e na aba *Activation* escolha *Minimum Pool Size=10* e *Max Pool Size=20* (**Figura 16**). Com isso estamos definindo que nosso *pool* terá no mínimo 10 componentes (leia-se *Transactional DataModule*) ativos e pode crescer até 20 conforme demanda. O *CreationTimeOut* indica em milésimos de segundo quanto tempo um objeto deve permanecer no *pool* se não for usado (*timeout*). Incremente esse valor de acordo com sua necessidade.

Agora algumas alterações são necessárias no *Transactional DataModule*. Defina a propriedade *KeepConnection* como *False* e *Connected* do *SQLConnection* como *True*. Isso fará com que a conexão fique aberta após ser usada, e quando o objeto for devolvido para o *pool*, não será necessário reabri-la quando um novo cliente solicitar dados do banco de dados. Isso acrescenta um ganho de performance imenso, visto que o tempo necessário para se estabelecer uma conexão TCP/IP, autenticar o usuário, conectar, não será necessário.

Para ver o *Object Pooling* em ação, no catálogo do COM+, selecione o item *Components* e a seguir clique no botão *Status*. Veja (**Figura 17**) que temos 10 objetos em *pool* (é necessário executar o cliente).

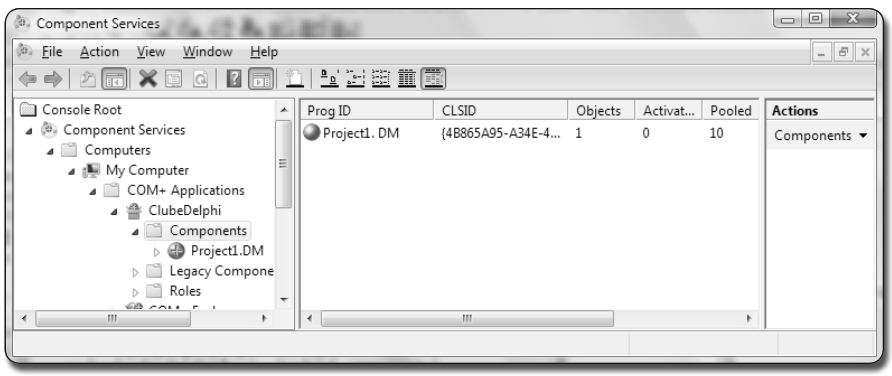


Figura 17. Object Pooling em ação

Listagem 3. Stored Procedure

```
create procedure AVG_SALARY (COD_DEPT integer)
returns (TOTAL numeric(15,2))
as
begin
   select
      avg (SALARY)
    from
       EMPLOYEE
    where
       DEPT_NO=:COD_DEPT
    into :TOTAL;
end
```

Listagem 4. Implementação do método remoto

```
function TDM.AVG_SALARY(DEPT_NO: Integer): Double;
begin
  SQLStoredProc1.ParamByName('COD_DEPT').AsInteger := DEPT_NO;
  SQLStoredProc1.ExecProc;
  result := SQLStoredProc1.ParamByName('TOTAL').AsFloat;
end;
```

Listagem 5. Chamando o método remoto

```
procedure TForm1.btAVGClick(Sender: TObject);
begin
 Label1.Caption := IntToStr(
   DM.DCOMConnection1.AppServer.AVG_SALARY
     (DM.cdsDepartmentDEPT_NO.AsInteger));
end;
```

SimpleObjectBroker

Um recurso interessante do *DataSnap* é a realização de um *balanceamento de carga*. Digamos que sua aplicação multicamadas, por algum motivo, comece a perder escalabilidade. Você pode adicionar mais servidores replicados e assim ter múltiplos servidores de aplicação na mesma solução. No cliente, usamos um componente chamado *SimpleObjectBroker* para aleatoriamente selecionar um objeto COM+ a ser usado em uma operação.

> **Nota**: O catálogo do COM+ permite criar um pacote de instalação MSI para você replicar automaticamente um servidor para outro.

Abra o *DM* da aplicação cliente e nele coloque um *SimpleObjectBroker* da paleta *DataSnap*. Defina sua propriedade *LoadBalanced* como *True*, abra o editor da propriedade *Servers* e adicione o nome dos computadores servidores (**Figura 18**). Agora defina a propriedade *ObjectBroker* do *DCOMConnection* para *SimpleObjectBroker1*.

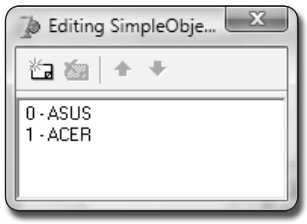


Figura 18. Balanceamento de carga no DataSnap

Chamando métodos remotos

IAppServer já nos quebra um imenso galho ao permitir realizar todas as operações de banco de dados em apenas 7 métodos. Porém, normalmente, teremos outros métodos em nossa interface, como métodos de negócio. Com isso, estamos estendendo a interface *IAppServer*, adicionando métodos personalizados.

Imagine o caso de chamar uma *Stored Procedure*. O cliente não pode invocar uma SP diretamente do banco de dados porque isso fere os princípios do desenvolvimento em 3 camadas. A comunicação deve passar pelo servidor de aplicação. Nesse caso, nada melhor do que criar um método personalizado no servidor de aplicação e invocá-lo a partir do cliente. Esse método no servidor é que fará a comunicação com o *Firebird*.

Consideremos o seguinte cenário. Uma SP que precisa retornar a média salarial de todos os funcionários de um determinado departamento do banco *Employee.fdb*. Usando o *IBExpert* ou a sua ferramenta preferida para administração de banco de dados *Firebird*, crie a *Stored Procedure* da **Listagem 3**, que retorna a média salarial de todos os funcionários de um dado departamento.

Abra o *Transactional DataModule* na aplicação servidora e coloque um *SQLStoredProc*. Aponte para o *SQLConnection* através da propriedade de mesmo nome. Em *StoredProcName* aponte para procedure *AVG_SALARY* criada anteriormente.

No IDE do *Delphi*, clique em *Tools>Environment Options*. Em *Type Library>Language* escolha *PASCAL*. Isso afetará como os tipos de dados serão exibidos no editor da *Type Library*.

Clique em *View>Type Library*, dê um clique de direita em *IDM* e escolha *New>Method*. Dê o nome de *AVG_SALARY* para ele. Na guia *Parameters* coloque *Double* em *Return Type* e adicione um parâmetro chamado *DEPT_NO (integer)*, como mostra a **Figura 19.**

Se você abrir o arquivo da *Type Library* que está no projeto, verá que nossa interface *IDM* define agora o *AVG_SALARY*:

```
IDM = interface(IAppServer)
  ['{59D9816F-5FB2-45C8-9593-33A37E4FD171}']
  function AVG_SALARY(DEPT_NO: Integer):
    Double; safecall;
end;
```

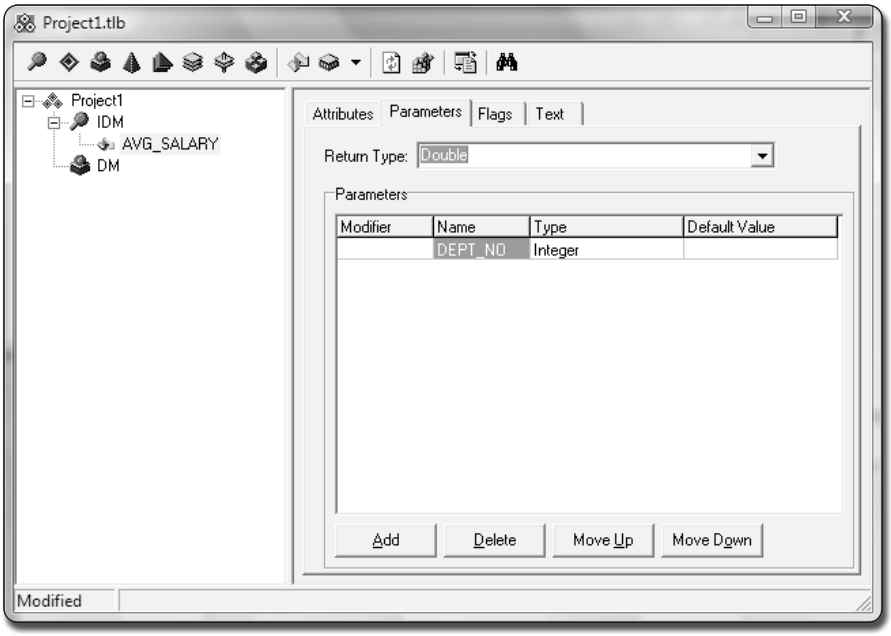


Figura 19. Método adicional na Type Library

Na *unit* do *Transactional DataModule* observe que o *Delphi* já declarou o método na *CoClass* e criou o cabeçalho da implementação. Resta-nos implementar então o método que chama a *Stored Procedure* no banco de dados e retornar o resultado para a aplicação cliente (**Listagem 4**).

Dê um *Build* no servidor. Na aplicação cliente coloque um botão em um *Label*. No evento *OnClick* do botão digite o código da **Listagem 5.** Execute a aplicação, selecione um departamento no *grid* e clique no botão para ver a média dos salários desse departamento (**Figura 20**).

Veja que usamos a propriedade *AppServer* do objeto de conexão *DataSnap* para invocar o método. *AppServer* é um tipo *variant*, o que significa que o compilador só vai reconhecer o verdadeiro significado das chamadas de métodos em tempo de execução, processo conhecido como *late-binding*. Veja por exemplo que podemos escrever qualquer coisa no lugar da chamada do método remoto, como mostra a **Listagem 6**, sendo que obteremos o erro em runtime da **Figura 21**. O efeito disso

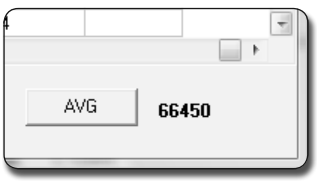


Figura 20. Chamando métodos remotos

Nota do DevMan

Type Library é um nome auto-explicativo. Ela é uma biblioteca (em duplo formato - .PAS e .TLB) que define os tipos de um objeto servidor, além de seus métodos. A *Type Library* funciona como o "controle remoto" em uma comunicação multicamadas, um protocolo. Ambos, cliente e servidor, devem conhecer e concordar com os teremos definidos na *Type Library*.

Nota do DevMan

Variants surgiram no Delphi 2 para suportar a automação OLE. São variáveis de tipo indefinido, que em teoria podem receber qualquer valor. São utilizados no *DataSnap* para prover o *late-binding* com o *AppServer*, ou seja, a ligação tardia, onde o compilador só irá identificar os métodos no momento da chamada.

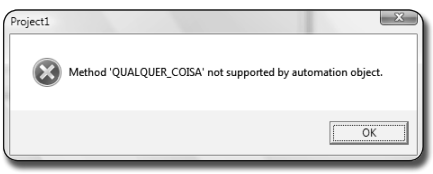


Figura 21. Late-binding pode gerar erros em runtime

Listagem 6. Chamando o método remoto

```
procedure TForm1.btAVGClick(Sender: TObject);
begin
 Label1.Caption := IntToStr(
   DM.DCOMConnection1.AppServer.QUALQUER_COISA
     (DM.cdsDepartmentDEPT_NO.AsInteger));
end;
```

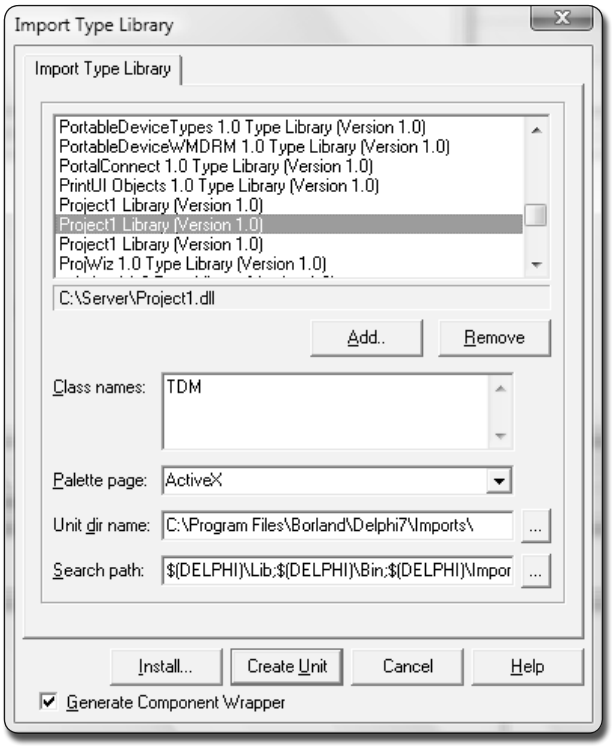


Figura 22. Importando a Type Library

Figura 23. Componente Wrapper

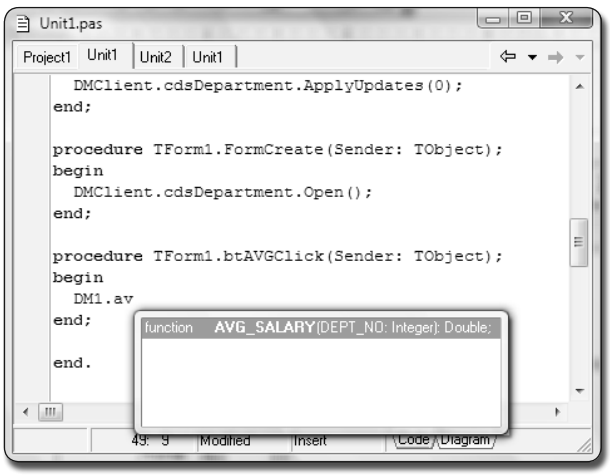


Figura 24. Early-Binding em ação

é que não temos suporte ao *Code Insight* ao chamar os métodos de *AppServer*.

Early Binding

Uma alternativa ao utilizar *late-binding* é usar *early-binding*, que faz uma ligação prévia com o servidor de aplicação e com isso o compilador é capaz de identificar os métodos em tempo de compilação, com suporte ao *Code Insight*. Isso é feito através da importação da *Type Library*. Na prática, isso é 5 vezes mais rápido do que usar *late-binding*.

No cliente clique em *Project>Import Type Library* (**Figura 22**). Deixe marcada a opção *Generate Component Wrapper*, o que fará com que o IDE crie um componente *Proxy* que encapsulará a *Type Library* para facilitar a comunicação com o *AppServer*. Clique em *Install, Yes* e *Ok*. Um componente (**Figura 23**) surgirá na paleta *ActiveX*.

> **Nota**: Renomeie neste momento o DM cliente para *DMClient* para não conflitar com o DM da biblioteca de tipos do componente servidor. Troque também as referências no código (eventos do botão *Apply* e *Form Create*).

Coloque no *form* o componente criado, que deverá se chamar *DM1*, representando o servidor de aplicação remoto. No botão que calcula a média, veja que agora podemos usar o *CodeInsight* para chamar a biblioteca de tipos (**Figura 24**). Digite o código da **Listagem 7** e faça o teste.

DataRequest

Uma outra alternativa a utilizar *Early-Binding* ou *Late-Binding* é utilizar a própria interface *IAppServer*, sem modificar a biblioteca de tipos. Isso pode economizar tempo, além de um novo registro da TLB caso o objeto já esteja instalado.

Para a comunicação de dados que não sejam necessariamente relacionados a *DataSets*, *IAppServer* oferece o método *DataRequest*, que na prática significa que você pode solicitar qualquer dado do *AppServer* e receber qualquer coisa como retorno.

No *AppServer*, no evento *OnDataRequest* do *DataSetProvider*, digite o código da **Listagem 8.** Note que o evento recebe

como parâmetro um *Input*, que pode ser algum comando vindo do cliente (que nesse caso vai ser o código do departamento). Recompile.

No cliente, no *OnClick* do botão AVG, chame o método *DataRequest* do *ClientDataSet* como mostra a **Listagem 9.** Execute a faça o teste. Note que não foi preciso modificar nem usar a biblioteca de tipos, pois tudo já é feito por *IAppServer*.

Executando como biblioteca

Imagine a seguinte situação. Você criou um poderoso sistema 3 camadas, com vários objetos COM+ e vários clientes, porém um usuário mais humilde resolveu comprar sua solução e vai rodá-la inteira numa simples máquina (nesse caso Banco de Dados + *AppServer* + cliente residiriam na mesma máquina).

Nessa situação, não haveria a necessidade de uma solução multicamadas, você poderia optar por usar *DataSnap* local. Mas graças ao COM+, temos uma elegante saída que pode otimizar sua aplicação para esse cenário.

Tal solução consiste em rodar o servidor como uma *biblioteca*, e não como um servidor. Nesse caso, quando o cliente ativar a comunicação com o *AppServer*, ao invés de ser levantado mais um processo *DLLHost.exe* para hospedar a DLL e conseqüentemente estabelecer um processo de comunicação *out-of-proc* (entre processos, envolvendo *marshalling* de dados), o servidor inteiro (DLL) é levantado no mesmo endereço de memória do processo chamador. Na prática, isso é como se servidor e cliente fossem compilados em um único EXE, pois a comunicação é *in-proc* (no mesmo processo), sem perda de desempenho e *marshalling*.

Para ver isso na prática, no gerenciador do COM+ dê um clique de direita no nosso aplicativo *ClubeDelphi*. Em *Activation*, marque a opção *Library Application* (lembre-se de voltar ao *default – server* – após o teste), como mostra a **Figura 25**. Agora execute o cliente. Em *View>Debug Windows>Modules*, note que são mostradas todas as DLLs que foram carregadas no mesmo endereço do processo EXE cliente, incluindo o próprio servidor *DataSnap* (**Figura 26**). Acredito que com isso você não tenha mais motivos para não migrar para *DataSnap* sua solução *client/server*.

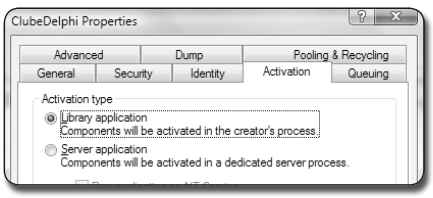
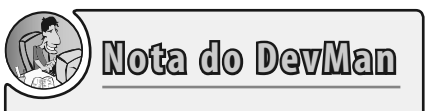


Figura 25. Opção Library Application



Nota do DevMan

Muito cuidado ao modificar interfaces COM. Como comentado anteriormente, interfaces são imutáveis, então planeje bem seu servidor de aplicação antes de implementá-lo. Normalmente uma mudança nos métodos do servidor requer um novo registro da biblioteca e uma nova geração da *Type Library*.

Nota do DevMan

A solução com *Library* torna o aplicativo duas camadas. Uma opção adicional seria rodar o *Firebird* como *Embedded*, aí o próprio BD rodaria no mesmo processo do cliente onde já está o servidor de aplicação, tudo em uma camada só. Para mais informações veja a minha aula em *http://www.devmedia.com.br/articles/viewcomp.asp?comp=1734*

Listagem 7. Chamando o método remoto

```
procedure TForm1.btAVGClick(Sender: TObject);
begin
  Label1.Caption:= FloatToStr(DM1.AVG_SALARY(
    DMClient.cdsDepartmentDEPT_NO.AsInteger));
end;
```

Listagem 8. OnDataRequest do DataSetProvider

```
function TDM.dspDepartmentDataRequest(Sender: TObject;
  Input: OleVariant): OleVariant;
begin
  SQLStoredProc1.ParamByName('COD_DEPT').AsInteger := Input;
  SQLStoredProc1.ExecProc;
  result := SQLStoredProc1.ParamByName('TOTAL').AsFloat;
end;
```

Listagem 9. DataRequest

```
procedure TForm1.btAVGClick(Sender: TObject);
begin
    Label1.Caption := DM.cdsDepartment.DataRequest(
       DM.cdsDepartmentDEPT_NO.AsInteger)
end;
```

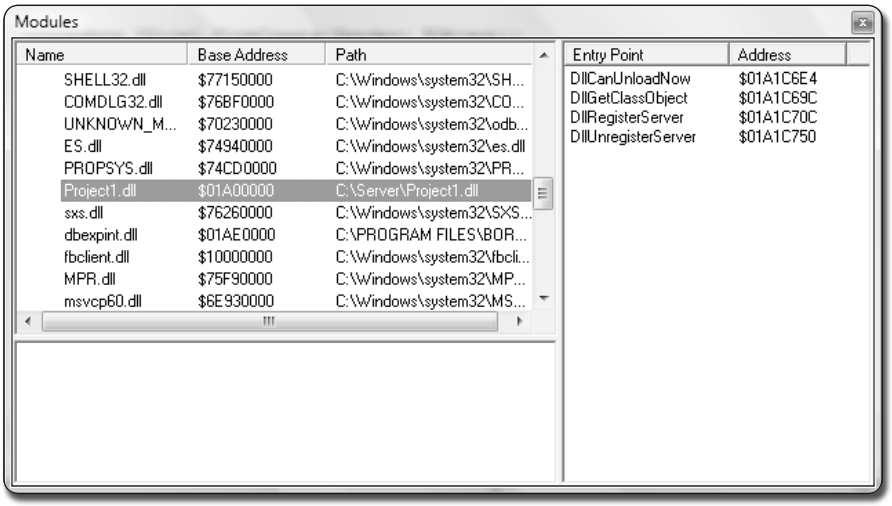


Figura 26. Servidor DataSnap inteiro rodando junto ao cliente

Debugando servidores

Um procedimento comum no desenvolvimento *DataSnap* consiste em debugar o servidor de aplicação. Mas como debugar se ativação é *Just-in-time* (na hora da chamada cliente)? Ou seja, não podemos executar o servidor pois é uma DLL, precisamos executar o cliente e debugar a DLL. Para isso, alguns passos são necessários. No servidor, clique em *Project>Options*. Na aba *Linker* marque as opções conforme a **Figura 27** (lembre-se de voltar ao padrão após o *debug* para não afetar a otimização da DLL). Na aba *Compiler* configure como mostra a **Figura 28.**

Agora, em *Run>Parameters*, configure como mostrado na **Figura 29**. Note que colocamos o *Host Application* como o processo *DLLHost*. Em *parameters*, colocamos o seguinte:

```
/Processid:{1DC8FE61-0CEB-498D-823A-C3C3C4169C7F}
```

Esse GUID é o do pacote servidor, e com certeza vai mudar na sua máquina. Então pegue-o no catálogo do COM+, dando um clique de direita no nome da aplicação e selecionando o *Application ID*. Execute o cliente e clique no botão AVG. A **Figura 30** mostra o exemplo sendo depurado.

Parâmetros

E finalmente, para concluir o exemplo, não poderíamos deixar de falar em parâmetros. No presente momento, estamos trazendo todos os departamentos e para cada um, todos os seus respectivos funcionários. Isso, sem dúvida, é muita informação para ser trafegada em um ambiente multicamadas. Que tal pedir ao cliente para que digite o código do departamento, e aí sim trazemos somente o departamento solicitado com seus respectivos funcionários?

O primeiro passo é trocar a instrução SQL do *qryDepartment* para:

```
select * from DEPARTMENT
where DEPT_NO=:DEPT_NO
```

Configure o parâmetro criado como sendo do tipo *Input* e *Integer*. Recompile o servidor de aplicação.

No cliente, coloque um *Edit* e um *Button* como mostra a **Figura 31.** No *ClientDataSet*, dê um clique de direita e escolha a opção *Fetch Params*, o que fará com que o parâmetro definido no *AppServer* venha parar na aplicação cliente e possa ser passado diretamente no *ClientDataSet* (**Figura 32**). O resto é por conta de *IAppServer*.

Retire o *Open* do *OnCreate* do *form* e no *OnClick* do novo botão digite o código da **Listagem 10**. Execute e digite um código de departamento no *Edit* (ex.: 600) e clique no botão. Veja que somente o departamento desejado é trafegado na rede (**Figura 33**), junto com seus respectivos funcionários através do *DataSetField*.

Deploy

Até agora, utilizamos a mesma máquina para ambos cliente e servidor, apesar de serem dois processos. Mas como fazer isso funcionar em uma *intranet* real, com duas ou mais máquinas físicas? Pois bem, aproveitei minha infra-estrutura de rede em meu escritório para realizar os testes. Ao executar a aplicação em outro terminal, obtivemos um erro, claro, pois o cliente não sabe onde está o servidor e nem como acessá-lo ("Classe não registrada"). Mas o procedimento é simples:

1 - No servidor, compartilhe o disco ou o diretório onde está a aplicação servidora (*Server*);

2 - Na máquina cliente, por compartilhamento de rede, mapeie o diretório/disco compartilhado no passo 1;

3 - Na máquina cliente, digite o seguinte no *prompt* para registrar a *Type Library*:

```
tregsvr z:\Server\Project1.dll
```

Listagem 10. Passando parâmetros via IAppServer

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  DMClient.cdsDepartment.Close();
  DMClient.cdsDepartment.Params[0].AsInteger :=
    StrToInt(Edit1.Text);
  DMClient.cdsDepartment.Open();
end;
```

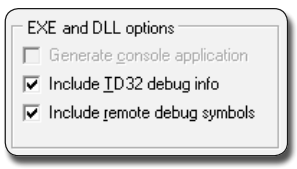


Figura 27. Opções para Debug

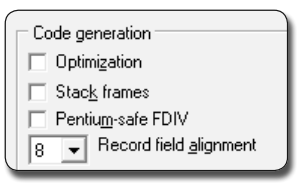


Figura 28. Mais opções para Debug

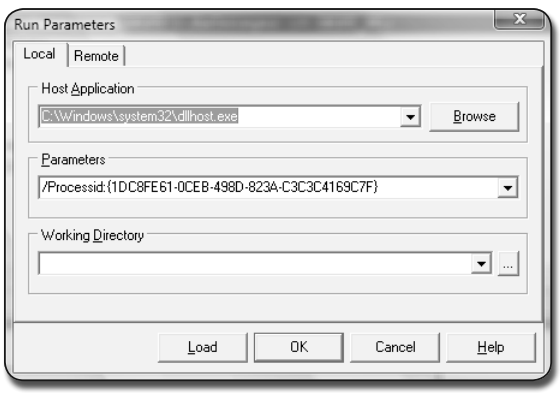


Figura 29. Run Parameters

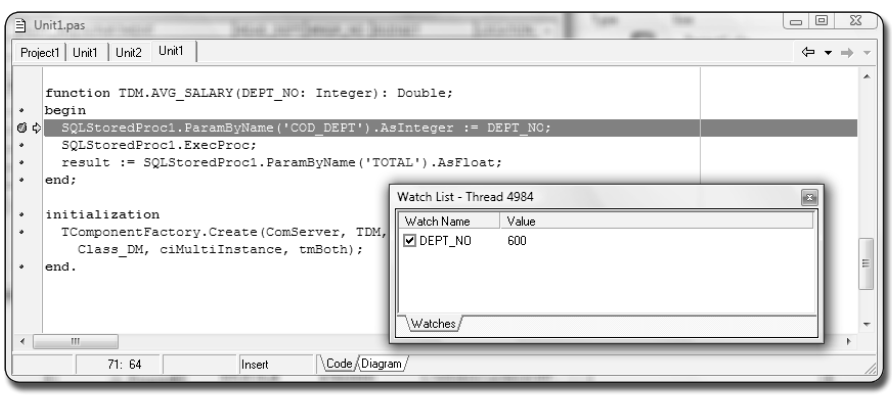


Figura 30. Depurando a aplicação servidora

Onde z é o disco onde está aplicação servidora e o restante o caminho da DLL. *TRegSvr* é um aplicativo que vem com o *Delphi* e pode ser copiado para o local onde você vai executar o comando no *prompt*.

Pronto! Temos o cliente acessando o servidor, através de uma LAN, e a chamada remota sendo executada perfeitamente.

Nota: Se não existir a biblioteca MIDAS.DLL no cliente, copie-a para essa estação (diretório *System* do *Windows*), ou declare *MidasLib* em uma unit qualquer da aplicação cliente, recompile e refaça o *deploy*.

Figura 31. Formulário pronto para passagem de parâmetro

Figura 32. Trazendo os parâmetros via IAppServer

Conclusão

O desenvolvimento *DataSnap* é elegante, rápido, produtivo e robusto. Sabendo aplicar bem as técnicas aqui demonstradas, vimos como criar uma solução escalável para um ambiente multicamadas. Acredito que não existam mais motivos para você não migrar mais sua solução *client/server* para um próximo nível em boas práticas de desenvolvimento. ●

Links

http://www.distribucon.com/

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

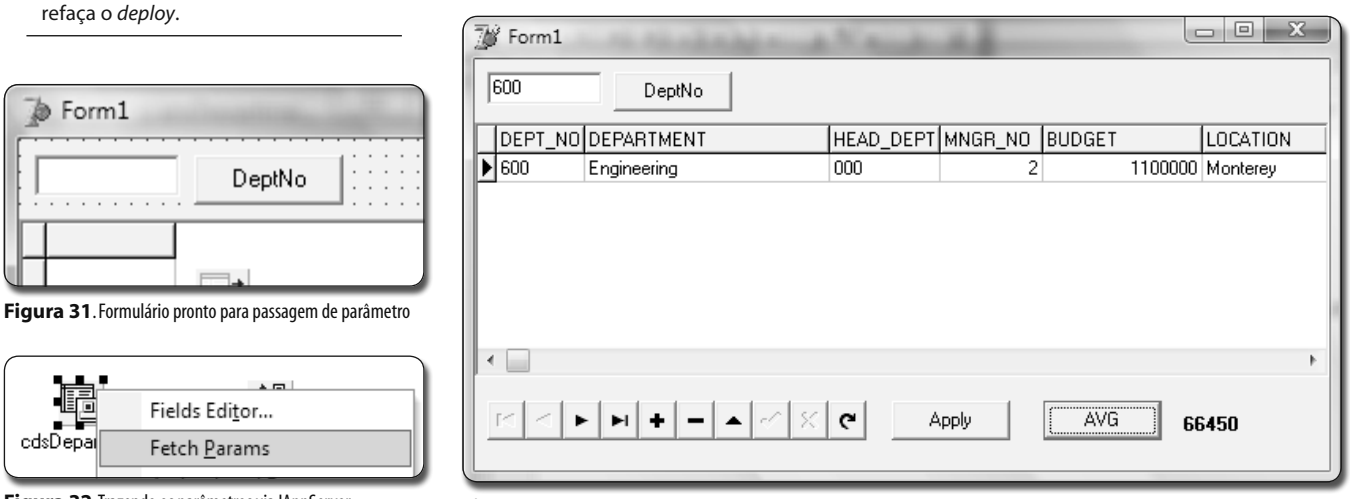
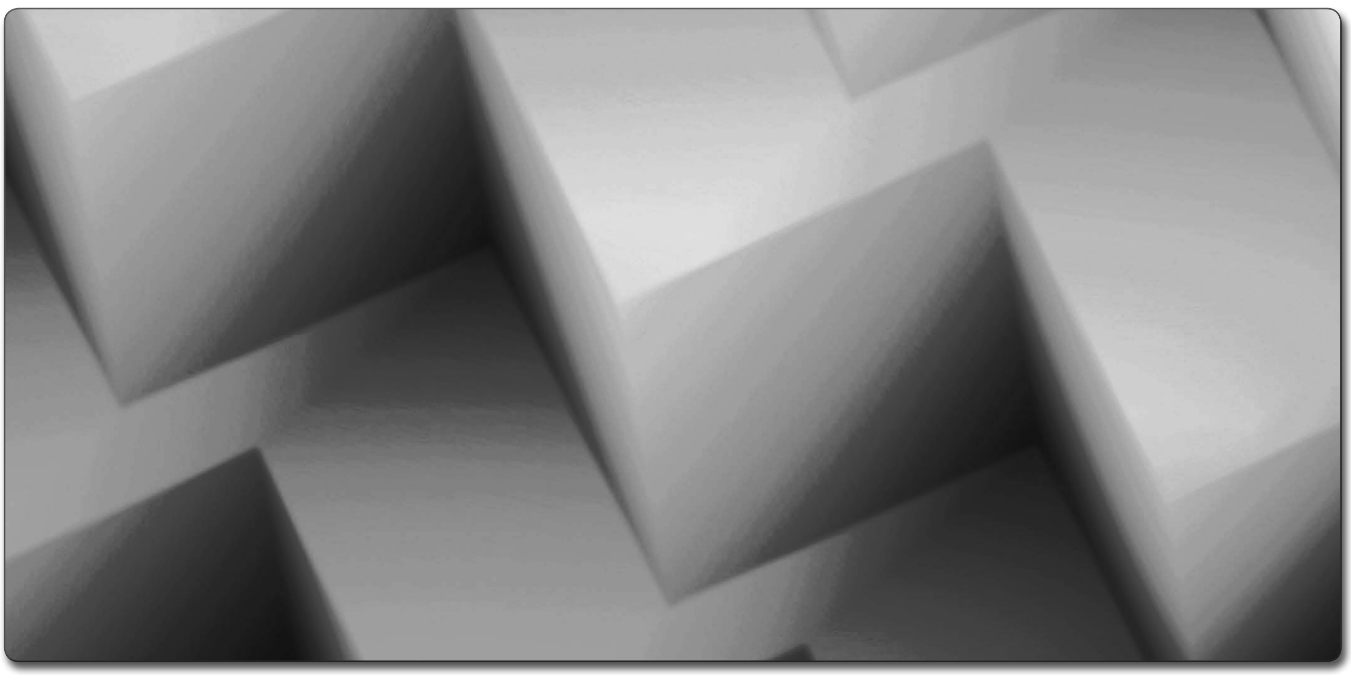
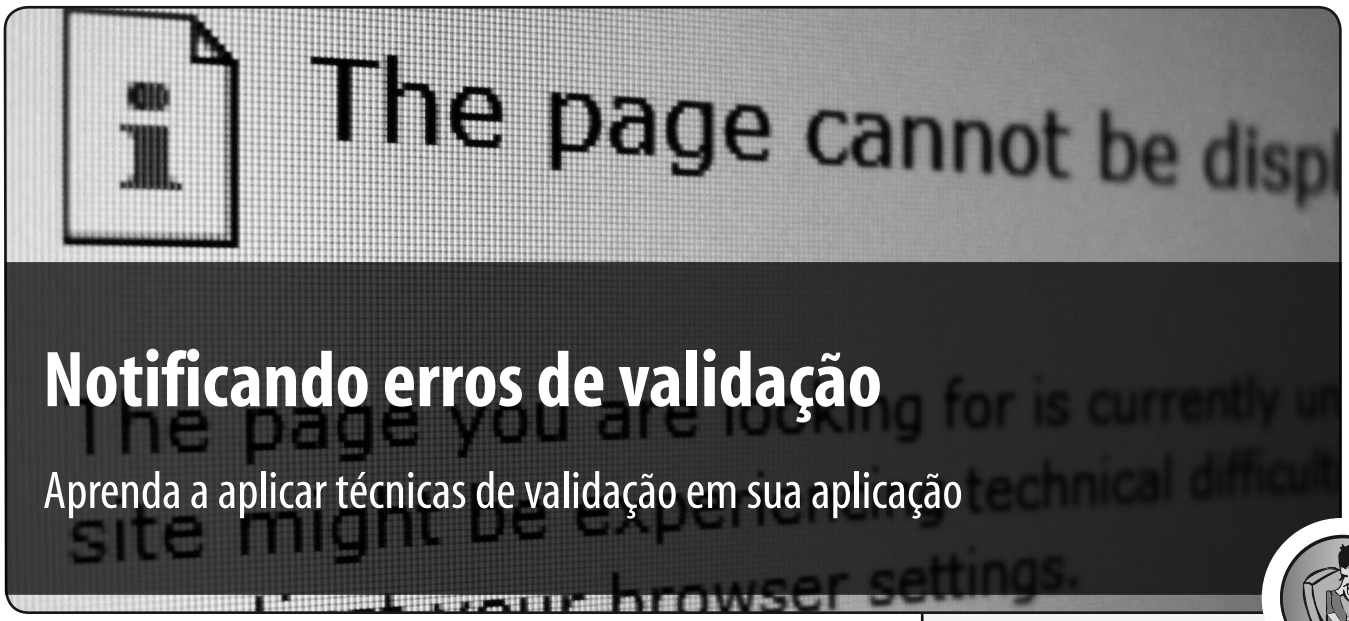


Figura 33. Opção Library Application



Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software



Notificando erros de validação

Aprenda a aplicar técnicas de validação em sua aplicação

Paulo Roberto Quicoli
(pauloquicoli@gmail.com)
é analista e programador da Control-M Informática. Trabalha com Delphi, desde sua primeira versão, e Firebird desenvolvendo aplicações cliente-servidor. Formado em Tecnologia de Processamento de dados pela FATEC, na cidade de Taquaritinga/SP. Blog: *http://pauloquicoli.spaces.live.com*

Resumo DevMan

Com cada vez mais aplicações no mercado, tornar nossa alternativa mais atraente e funcional passou a ser ainda mais importante. Alertar o usuário de possíveis erros é uma premissa importante no desenvolvimento de qualquer aplicação. Veremos nesse artigo como implementar algumas classes para a exibição de erros de forma funcional e visual.

Nesse artigo veremos

- Orientação a Objetos;
- Criação de Eventos;
- Criação de um notificador para nossas aplicações.

Qual a finalidade

- Desenvolver um pequeno framework que exiba de forma agradável ao usuário mensagens de validação.

Quais situações utilizam esses recursos?

- Formulários para entrada de dados.

Ao observarmos a evolução dos aplicativos *Windows*, incluindo o próprio sistema operacional, vemos uma dedicação especial dos desenvolvedores para os recursos visuais. Por recursos visuais quero dizer a própria aparência em si e também a experiência do usuário no que diz respeito à usabilidade. Porém, nem só de visual vive uma aplicação. Seja *Win32* ou *Web*, uma aplicação precisa prever possíveis erros do usuário e notificá-lo, como um *CPF* ou *CNPJ* incorretos. Dessa forma teremos sempre um sistema visualmente atraente e ao mesmo tempo inteligente a ponto de prever erros e/ou falta de parâmetros em determinadas partes do sistema.

Neste artigo vamos desenvolver um pequeno *framework* de notificação de erros que tem como objetivo melhorar essa experiência do usuário, provando que um bom sistema precisa ir muito mais além do que o visual.

Funcionamento e estrutura básica

Para desenvolver o exemplo, tomei idéias do validador do *.net* e também do *JVCL Validator*. O objetivo então é informar que um determinado controle visual – *TEdit, TDBEdit* – possui um valor não válido, seja mudando o controle de cor, movimentando o controle ou incluindo um ícone de alerta ao lado do mesmo. Também deve permitir obter, em forma de lista, todos os controles que não passaram nas validações, porém o código de validação em si não é de responsabilidade do *framework*, ou seja, é necessário apenas informar que um determinado controle está *ok* ou não. Na **Figura 1** vemos como nosso *ErrorNotifier* está dividido.

O *ErrorNotifier* é dividido em duas partes. O centro de notificação, que armazena os controles inválidos é um notificador, que é responsável por realizar a real notificação. Dessa forma, estamos aplicando o conceito de *responsabilidade de objetos*", onde não se deve acumular várias rotinas em uma mesma classe e sim dividi-las em classes, distribuindo a responsabilidade.

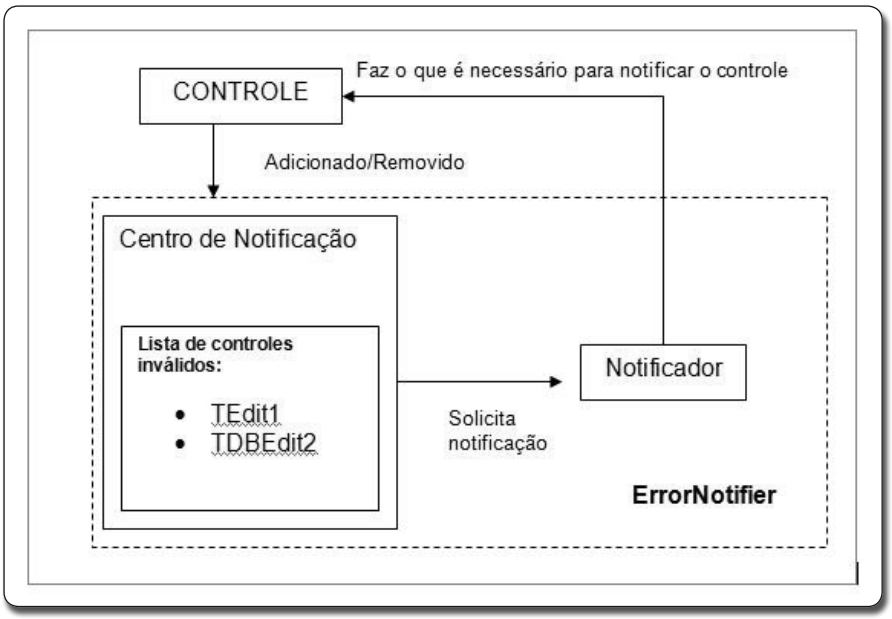


Figura 1. Funcionamento do ErrorNotifier

Desenvolvimento

Aqui utilizei o Delphi 7, sinta-se à vontade para utilizar a versão que possuir. Inicie um novo projeto usando o menu *File>New>Application*, salve-o com o nome que preferir e em seguida crie uma nova *unit* usando o menu *File>New>Unit* salvando-a como *ErrorNotifier.pas*. Declare as seguintes classes:

```
TErrorNotifierCenter = class;
TInvalidControl = class;
THackWinControl = class(TWinControl);
```

Aqui temos nosso centro de notificação, representado pela classe *TErrorNotifierCenter*. Nele vamos adicionar os controles inválidos, que estão representados pela classe *TInvalidControl*. Essa classe irá conter algumas informações extras que serão utilizadas pelo *Notificador*, como a mensagem de erro a ser exibida, o controle que está inválido e o *hint* atual do controle. Para que possamos notificar vários tipos de controles, *TInvalidControl* não irá conter diretamente um controle do tipo *TEdit* ou *TDBEdit*, e sim um *TWinControl*, que está na hierarquia da maioria dos controles visuais de edição de dados. Definimos também a classe *THackWinControl* para que tenhamos acesso a alguns métodos protegidos de *TWinControl*. Implemente *TInvalidControl* como na **Listagem 1**.

Listagem 1. Código da classe TInvalidControl

```
TInvalidControl = class
private
  FAlertHint: string;
  FOriginalHint: string;
  FControl: TWinControl;
  procedure SetAlertHint(const Value: string);
  procedure SetControl(const Value: TWinControl);
  procedure SetOriginalHint(const Value: string);
public
  constructor Create(pControl: TWinControl; sOriginalHint, sAlertHint: string);
  property Control: TWinControl read FControl write SetControl;
  property OriginalHint: string read FOriginalHint write SetOriginalHint;
  property AlertHint: string read FAlertHint write SetAlertHint;
end;
```

Observe que temos apenas propriedades nessa classe, portando não vou listar aqui seu código, visto que são apenas *Get's* e *Set's*.

O Notificador

Para permitir flexibilidade vamos criar uma interface que deverá ser obedecida na implementação de qualquer *Notificador*, ou seja, ao final do *framework* qualquer desenvolvedor poderá criar uma nova forma de notificação, desde que siga a interface especificada. A seguir veja a declaração de nossa *interface*.

```
INotifier = interface
  ['{D76BA182-9E19-48E6-86CE-5EC69DCF7D8D}']
  procedure StartNotifying(pInvalidControl:
    TInvalidControl);
  procedure StopNotifying(pInvalidControl:
    TInvalidControl);
end;
```

Os dois métodos definidos recebem como parâmetro um objeto do tipo *TInvalidControl* e é através desse objeto que o controle será acessado e manipulado.

> **Nota:** Em *Win32* toda interface necessita de um identificador único. Esse identificador é o *GUID*, o código entre colchetes. Ele não pode ser qualquer número, deve ser gerado. No Delphi basta pressionar *CTRL+G* e o *GUID* será gerado.

Centro de notificação

Este é o núcleo do *framework*. É quem mantém a lista de controles inválidos e dispara a ordem de notificação. Na **Listagem 2** temos a estrutura da classe *TErrorNotifierCenter* e na seqüência os detalhes sobre seus métodos.

No *create* de nossa classe passamos um objeto que implementa *INotifier*, ou seja,

Listagem 2. Estrutura da classe TErrorNotifierCenter

```
TErrorNotifierCenter = class
private
  FNotifier: INotifier;
  FInvalidControlsList: TObjectList;
  FErrorList: TStringList;
  function ControlInList(pControl: TWinControl): integer;
  function GetFirstErroneousControl: TWinControl;
  function GetHasErros: boolean;
  function getErrorList: TStringList;
public
  constructor Create(Notifier: INotifier); reintroduce;
  destructor Destroy; override;
  procedure AddInvalidControl(pControl: TWinControl; AlertMessage: string);
  procedure RemoveInvalidControl(pControl: TWinControl);
  property ErrorList: TStringList read getErrorList;
  property HasErrors: boolean read GetHasErros;
  property FirstErroneousControl: TWinControl read GetFirstErroneousControl;
end;
```

Listagem 3. Adicionando um controle inválido

```
procedure TErrorNotifierCenter.AddInvalidControl(pControl: TWinControl;
  AlertMessage: string);
var
 index: integer;
begin
  if ControlInList(pControl) < 0 then
  begin
    Index := FInvalidControlsList.Add(TInvalidControl.Create(
      pControl,pControl.Hint, AlertMessage));
    FNotifier.StartNotifying(FInvalidControlsList[index] as TInvalidControl);
  end;
end;
```

Listagem 4. Procedimento RemoveInvalidControl

```
procedure TErrorNotifierCenter.RemoveInvalidControl(pControl: TWinControl);
var
 Index: integer;
begin
  Index := ControlInList(pControl);
  if Index >= 0 then
  begin
    FNotifier.StopNotifying(FInvalidControlsList[index] as TInvalidControl);
    FInvalidControlsList.Delete(index);
  end;
end;
```

Listagem 5. TdefaultNotifier

```
TDefaultNotifier = class(TInterfacedObject, INotifier)
private
  FDefaultColor: TColor;
  FErrorColor: TColor;
public
  constructor Create(DefaultColor, ErrorColor: TColor);
  procedure StartNotifying(pInvalidControl: TInvalidControl); virtual;
  procedure StopNotifying(pInvalidControl: TInvalidControl); virtual;
end;
```

Listagem 6. Implementação das procedure StartNotifying e StopNotifying

```
procedure TDefaultNotifier.StartNotifying(pInvalidControl: TInvalidControl);
begin
  THackWinControl(pInvalidControl.Control).Color := FErrorColor;
  pInvalidControl.Control.Hint := pInvalidControl.AlertHint;
end;
procedure TDefaultNotifier.StopNotifying(pInvalidControl: TInvalidControl);
begin
  THackWinControl(pInvalidControl.Control).Color := FDefaultColor;
  pInvalidControl.Control.Hint := pInvalidControl.OriginalHint;
end;
```

TErrorNotifierCenter conhecerá apenas os métodos definidos pela interface *INotifier*, dessa maneira podemos criar o *Notificador* que desejarmos, sem contudo, alterar uma linha de código no *ErrorNotifierCenter*. Neste construtor inicializamos os campos *FNotifier, FInvalidControlList* (*Lista de controles inválidos*) e *FErrorList* (*Lista com os erros existentes*).

Em seguida, declaramos uma *procedure* chamada *AddInvalidControl* como podemos ver a seguir:

```
procedure AddInvalidControl(pControl:
  TWinControl; AlertMessage: string);
```

Este é o método que será usado em nossos programas para *dizer* ao *framework* que um determinado controle está inválido. Passamos como parâmetro o controle e a mensagem de erro. Na **Listagem 3** temos sua implementação.

Aqui um objeto do tipo *TInvalidControl* é instanciado e armazenado na lista de controles inválidos *FInvalidControlsList* e logo após é solicitado ao *Notificador*, aquele passado como parâmetro do *Create*, que inicia a notificação do controle adicionado. Quero ressaltar aqui algo muito importante: veja que não criamos dependência entre quem mantém a lista dos controles e quem realiza a notificação. *TErrorNotifierCenter* não sabe, e nem precisa saber, como a notificação será realizada, ele apenas ordena: *Notificador, notifique o controle agora!*. Para que não corramos o risco de adicionar o mesmo controle duas vezes o método *ControlInList* é chamado.

Mais uma função é declarada em seguida, a *ControlInList*, como mencionado anteriormente. Veja sua declaração:

```
function ControlInList(pControl:
  TWinControl): integer;
```

Ela verifica se o controle passado como parâmetro já foi adicionado. Caso já tenha sido adicionado sua posição na lista é retornada. Essa verificação é feita utilizando um laço *for*, varrendo a lista e comparando a propriedade *name* dos controles.

Outro procedimento importante é o *RemoveInvalidControl:*

```
procedure RemoveInvalidControl(pControl:
  TWinControl);
```

Este é o método que será usado em nossos programas para *dizer* ao *framework* que um determinado controle está válido e deverá ser removido da lista. Na **Listagem 4** temos sua implementação.

Para também evitar que o usuário, por engano, tente retirar da lista um controle que não está lá, chamamos o método *ControlInList*.

```
function TErrorNotifierCenter.getErrorList:
  TStringList;
```

Essa função retorna uma lista com as mensagens de erro de cada controle existente na lista de controles inválidos. Essa lista pode posteriormente ser utilizada em um *ListBox* como uma espécie de log dos erros encontrados.

Criando um notificador

Como mencionado anteriormente um *notificador* deve implementar a interface *INotifier*. Nosso *notificador* irá deixar o controle em vermelho caso esteja inválido. Adicione uma nova classe na *Unit ErrorNotifier* e a defina como na **Listagem 5**.

No *Create* passamos duas cores que representam o estado do controle. *DefaultColor* quando o controle é válido e *ErrorColor* quando está inválido. Para saber como isso é feito, temos que observar os métodos *StartNotifying* e *StopNotifying* na **Listagem 6**.

Para acessarmos a propriedade *Color*, que é uma propriedade protegida, usamos nossa classe de *Hack* e então passamos a cor apropriada.

Colocando pra funcionar

Nesse momento, já temos nossas classes todas criadas e funcionando. Vamos ver agora como criar um aplicação e utilizá-las. Altere o formulário principal do projeto para que fique com a aparência semelhante à **Figura 2** e adicione a *Unit ErrorNotifier* na seção *Uses*. Se preferir utilizar o atalho *Alt + F11* e escolha a *unit ErrorNotifier*.

Na seção *private* do formulário adicione um campo chamado *FCentralErros* do tipo *TErrorNotifierCenter*. Este campo será nossa referência ao *TErrorNotifierCenter*. No *create* do formulário vamos então instanciá-lo chamando o método *Create* da classe *TErrorNotifierCenter*, basta fazer o seguinte:

Nota do DevMan

Quando utilizada de forma correta, as *Interfaces* realizam uma operação de *Garbage Collection*, nos livrando assim da necessidade de se dar um *Free* na referência. Por isso não criamos uma variável do tipo *TDefaulNotifier*. Veja mais detalhes sobre o uso correto de interfaces em: *dn.codegear.com/article/30125*

```
FCentralErros := TErrorNotifierCenter.Create(
  TDefaultNotifier.Create(clWhite, clRed));
```

Veja que no próprio *Create* do *TErrorNotifierCenter* estamos instanciando um *INotifier*. Podemos fazer isso sem nos preocupar com *memory leaks* porque *TErrorNotifierCenter* nesse momento espera uma implementação da interface *INotifier*, veja **Listagem 3**.

Agora é preciso informar ao *framework* que algum controle está inválido. Então no *onChange* do primeiro *Edit* vamos verificar se o conteúdo é vazio, se for, o controle está inválido. Veja como fazemos isso a seguir. Execute a aplicação e veja!

```
if Trim(editNaoVazio.Text) = '' then
  FCentralErros.AddInvalidControl(
    editNaoVazio,'Campo não pode ser vazio')
else
  FCentralErros.RemoveInvalidControl(
    editNaoVazio);
```

Aprimorando o TErrorNotifierCenter

Imagine que é necessário exibir a lista de erros que existem, como um resumo, como podemos fazer isso? O *framework* expõe o método *getErrorList* que retorna essa lista, porém fica a cargo do usuário chamar o método sempre que necessário. Porém podemos melhorar isso através do uso de eventos. Sempre que um controle for removido ou incluído disparamos dois eventos, um avisando que a lista foi atualizada e outro informando detalhes do controle. Vamos implementar a primeira situação.

Quando o evento a ser criado apenas retorna o *Sender*, por exemplo o clique de um botão, é preciso criar um tipo que herde de *TNotifyEvent*. *TNotifyEvent* é o evento básico do Delphi. Para ser caracterizado como evento, um tipo deve ser uma *procedure (Sender: TObject) of object*, e *TNotityEvent* representa esse tipo básico. Adicione então o seguinte tipo na seção *type* da *Unit ErrorNotifier*:

Nota do DevMan

Sempre que você for criar um evento personalizado, crie o tipo adicionando a palavra *Event* a ele. E na propriedade que dará o acesso ao tipo, adicione o prefixo *On*. Dessa forma mantemos o padrão do próprio Delphi, por exemplo o evento *OnKeyUp* de um *TEdit* é um evento do tipo *TKeyEvent*.

```
TErrorListUpdatedEvent = TNotifyEvent;
```

Agora, como adicionamos esse evento em nossa classe? A resposta é muito simples. Crie uma propriedade para ela:

```
property OnErrorListUpdated:
  TErrorListUpdatedEvent read
    FOnErrorListUpdated write
    FOnErrorListUpdated;
```

Já foi dito que um evento é uma *procedure*. Portanto para dispará-lo basta chamá-lo como uma *procedure* comum. Veja na **Listagem 7** como ficaram os métodos *RemoveInvalidControl* e *AddInvalidControl*.

Quando você efetua um clique duplo em um botão, por exemplo, em seu código fonte é criada uma *procedure*, como a seguir:

```
procedure TForm1.Button1Click(sender: TObject)
```

Então você codifica ali o que é necessário. Agora você sabe como o Delphi sabe que precisa, no momento do clique duplo no botão, chamar o seu código? Muito bem, um botão também possui uma propriedade do tipo do evento, e o IDE do Delphi se encarrega de gerar, no arquivo *.dfm*, a ligação da propriedade que representa o evento com o seu código. Essa ligação fica em um campo privado. Porém, e se não existe código associado a um evento, como não dispará-lo? É preciso então verificar se o campo que armazena a referência do evento está nulo, isso é feito através da função *Assigned*. Veja que antes de chamar o evento, nós verificamos sua situação.

Implementando o evento

Em nosso formulário principal podemos agora implementar o evento criando um método que possua a *mesma assinatura* do evento e então associá-lo à propriedade da classe. Veja como fazemos isso na **Listagem 8**.

Execute a aplicação e você verá que a lista de erros agora é atualizada automaticamente.

Criando seu próprio notificador

Como já foi mencionado, basta criar uma nova classe que implemente a interface *INotifier*. Essa nova classe necessariamente não precisa estar na *Unit ErrorNotifier*, você pode criar uma nova *Unit* e adicionar ao *Uses* a referência à *ErrorNotifier*. Junto aos arquivos deste projeto existe também uma *Unit ImageNotifier.pas*. Ela é uma implementação de notificador que exibe um ícone de erro ao lado de cada controle. Na **Figura 3** o vemos em execução.

Conclusão

De forma simples podemos melhorar o *relacionamento* de nossos clientes com nossos sistemas. São pequenas coisas, que ao fim das contas, fazem diferença. Acredito que o usuário quer algo que o atenda, não apenas funcionalmente – que é o principal – mas também visualmente. Acredito que o visual de um sistema pode ser sim uma questão de desempate, já que ele é o que o usuário vê diariamente. Com um pouco de criatividade, e com o auxílio da orientação a objetos desenvolvemos um controle de notificação de erros reutilizável e *customizável*.

Até a próxima, e sejam criativos! ●

Listagem 7. Métodos chamando o evento

```
procedure TErrorNotifierCenter.AddInvalidControl(pControl: TWinControl;
  AlertMessage: string);
var
 index: integer;
begin
  if ControlInList(pControl) < 0 then
  begin
    index := FInvalidControlsList.Add(TInvalidControl.Create(
      pControl,pControl.Hint, AlertMessage));
    FNotifier.StartNotifying(FInvalidControlsList[index] as
      TInvalidControl);
    if Assigned(FOnErrorListUpdated) then
      OnErrorListUpdated(Self);
  end;
end;
procedure TErrorNotifierCenter.RemoveInvalidControl(pControl: TWinControl);
var
 index: integer;
begin
  index := ControlInList(pControl);
  if index >= 0 then
  begin
    FNotifier.StopNotifying(FInvalidControlsList[index] as
      TInvalidControl);
    FInvalidControlsList.Delete(index);
    if Assigned(FOnErrorListUpdated) then
      OnErrorListUpdated(Self);
  end;
end;
```

Listagem 8. Implementando um evento

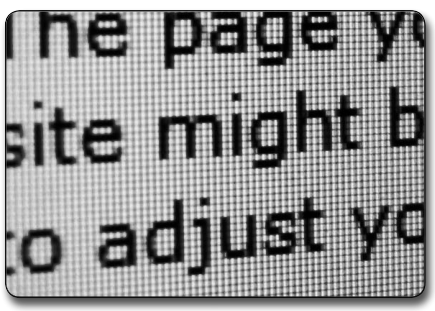
```
procedure TForm1.AtualizaLista(Sender: TObject);
begin
  ListBoxErros.Clear;
  ListBoxErros.Items.Assign(FCentralErros.ErrorList);
  btConfirmar.Enabled := not FCentralErros.HasErrors;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  FCentralErros := TErrorNotifierCenter.Create(TDefaultNotifier.Create(
    clWhite, clRed));
  FCentralErros.OnErrorListUpdated := self.AtualizaLista;
end;
```

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback



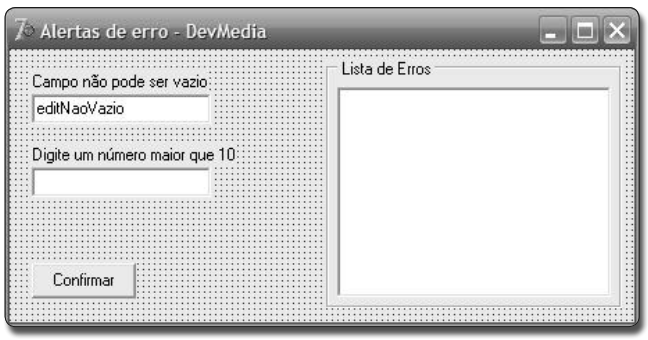


Figura 2. Exemplo de formulário para testes

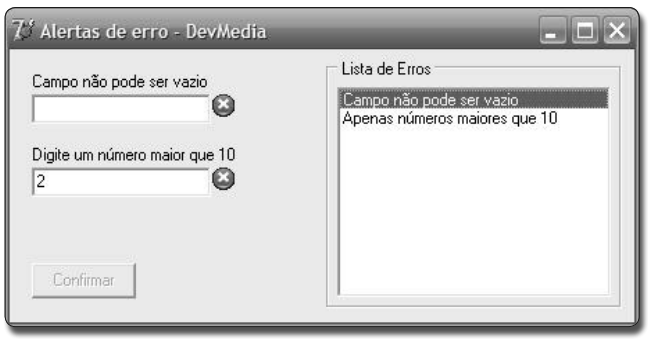


Figura 3. TimageNotifier em execução

Já pensou em hospedar todos os seus sites em um único plano de hospedagem?

PLANO PROFISSIONAL 1

3 domínios independentes
50 GB de transferência
30 GB de e-mails com SSL
1 GB de espaço
Painel de controle
ASP, ASP.NET 3.5 E PHP 5
Access ilimitado
3 bancos MYSQL 5
1 banco SQL Server 2005 Express

30 DIAS GRÁTIS

Ao assinar, digite o código de desconto exclusivo para leitores desta revista e ganhe!

RVSTMEDIA

Tudo isso por apenas R$ **25,90** por mês

Na Hospedix você ainda ganha registro de domínio (.com .net .org ou .info) grátis e isento de taxas anuais enquanto for cliente.

Acesse agora mesmo e descubra por que Hospedix é a hospedagem de sites preferida entre os profissionais.

hospedix.com.br

HOSPEDIX
HOSPEDAGEM PROFISSIONAL

Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software

Reaproveitamento de código

Organizando o código comum da aplicação

Resumo DevMan

O maior trauma de um desenvolvedor certamente é ter que desenvolver a mesma rotina mais de uma vez para a mesma aplicação ou para sistemas diferentes. Pensando nisso, diversos programadores optam por criar seus próprios componente e/ou bibliotecas de funções tais como: units personalizadas ou arquivos DLL.

Nesse artigo veremos algumas boas práticas no desenvolvimento de sistemas evitando a redigitação de código-fonte.

Nesse artigo veremos

• Técnicas de reaproveitamento de código;

• Criação de funções para uso em diversas partes do sistema;

Qual a finalidade

• Mostrar algumas das principais técnicas para evitar a digitação repetitiva de código pela aplicação;

Quais situações utilizam esses recursos?

• Em praticamente todo tipo de aplicação/programa podemos aplicar essas técnicas que facilitam o trabalho na hora da manutenção;

Luciano Pimenta
(lucianoalmeidapimenta@gmail.com)
é Técnico em Processamento de Dados, desenvolvedor Delphi/C#. Palestrante da 4ª edição da Borland Conference (BorCon). Autor de mais de 50 artigos e de mais de 270 vídeo aulas publicadas em revistas e sites especializados. Atualmente é programador Web da FullSoft - Mobile Solutions em Caxias do Sul-RS. Blog: lucianopimenta.blogspot.com. Site: www.lucianopimenta.net. É consultor da FP2 Tecnologia (www.fp2.com.br).

Nenhum programador gosta de refazer trabalho, e isso é bastante comum quando duplicamos código na aplicação. Imagine por exemplo, que temos um sistema que realiza baixa de contas a receber com lançamento no Caixa. O código para lançar no caixa está presente quando efetuamos a baixa na parcela e o mesmo pode estar quando do lançamento de uma conta, pois a mesma pode ter sido paga. Neste exemplo, teremos código duplicado na aplicação, onde o ideal é criar um método que recebe as informações do lançamento como parâmetro e executar apenas um código.

Neste artigo, vamos conhecer alguns métodos e funções que podemos organizar em uma *Lib* (biblioteca) para que possamos usar em todos nossos projetos.

Código Win32

Vamos mostrar nessa seção, códigos para aplicações *Win32*. Todos os exemplos mostrados aqui, podem ser coloca-

dos em uma *unit* e usado em qualquer aplicação *Win32*.

Um exemplo simples que sintetiza bem o reaproveitamento de código é a herança visual de formulários, onde é criado um formulário base, com código comum e não precisamos redeclarar o mesmo em cada formulário. Temos vários exemplos de uso de herança visual, então não mostrarei aqui como fazer essa técnica.

O primeiro exemplo de reaproveitamento de código é bastante usado pelos desenvolvedores, a criação de formulários. O código mais comum de usar para criação de um formulário é o seguinte:

```
Application.CreateForm(TfrmClientes,
frmClientes);
try
  frmClientes.ShowModal;
finally
  frmClientes.Release;
end;
```

Uma aplicação pode ter vários formulários, imagine esse código duplicado para cada formulário, sendo que a diferença entre eles é somente o formulário que será aberto. Para criar um método comum de criação de formulário, use o código da **Listagem 1**.

Agora, para chamar qualquer formulário, vamos usar apenas uma linha de código:

```
AbreFormulario(TfrmCliente, frmCliente);
```

Simples e prático. Claro, se você não quiser abrir o formulário com o *ShowModal*, deverá adaptar o código.

Auto-incremento

Quem utiliza *InterBase*/*Firebird*, sabe que não temos campos auto-incremento no banco. Para simular isso, podemos usar *Generator* ou na versão mais nova, o *Sequence*. Muitos desenvolvedores, por diversas razões, necessitam do valor do código atual quando salvar o registro na aplicação Delphi.

Para ter esse valor atualizado, precisamos refazer a consulta ao banco e atualizar os dados do registro em tela. Nem sempre isso é a melhor maneira, portanto podemos ter um código que pega o valor do *Generator* no banco. Use o código da **Listagem 2** para isso.

A função que, retorna um inteiro, recebe como parâmetro o nome do *Generator* e o *SQLConnection*, componente para conexão com o banco. Você pode alterar esse parâmetro para o componente que esteja usando. Por fim, temos um parâmetro que indica se vamos incrementar o *Generator* (1 ou 0).

Assim, podemos verificar o valor do *Generator* e incrementar o mesmo, ou apenas saber o valor atual do mesmo. Com esse código, basta usar o mesmo no evento *OnNewRecord* do *DataSet*, indicando o seu valor para o *TField*.

Habilitar/Desabilitar controles de tela

Outro código bastante usado em aplicações é para habilitar/desabilitar controles de tela. O código é simples conforme temos na **Listagem 3**.

Fizemos um laço pelos controles em tela e verificamos se o mesmo descende de *TControl*, assim repassamos para a propriedade *Enabled* (usando *cast*) o valor passado como parâmetro (*True* ou *False*). Caso deseje verificar algum tipo específico, basta adicionar mais um *if*.

Listagem 1. Código para abrir o formulário

```
procedure AbreFormulario(aClasseForm: TComponentClass;
  aForm: TForm);
begin
  Application.CreateForm(aClasseForm, aForm);
  try
    aForm.ShowModal;
  finally
    aForm.Release;
  end;
end;
```

Listagem 2. Pegando o valor do Generator

```
function GeneratorID (aName: string;
  Connection: TSQLConnection;
  Incrementa: Boolean): integer;
var
  Qry: TSQLQuery;
begin
  Qry := TSQLQuery.Create(nil);
  try
    Qry.SQLConnection := Connection;
    if Incrementa then
      Qry.SQL.Add(
        'SELECT GEN_ID('+aName+', 1) FROM RDB$DATABASE')
    else
      Qry.SQL.Add(
        'SELECT GEN_ID('+aName+', 0) FROM RDB$DATABASE');
    Qry.Open;
    Result := Qry.Fields[0].AsInteger;
  finally
    FreeAndNil(Qry);
  end;
end;
```

Listagem 3. Habilitando/desabilitando controles de tela

```
procedure HabilitaDesabilitaControls(Value: Boolean);
var
  i : integer;
begin
  for i := 0 to ComponentCount -1 do
    if (Components[i] is TControl) then
       (Components[i] as TControl).Enabled := Value;
end;
```

Nota: O código é mais utilizado em formulários, mas caso deseje implementar o código em um *unit* em separado e apenas chamar o mesmo em cada formulário, basta adicionar mais um parâmetro do tipo *TForm* para poder utilizar o *ComponentCount*.

ClubeDelphi PLUS www.devmedia.com.br/clubedelphi/portal.asp

Acesse agora o mesmo o portal do assinante ClubeDelphi e assista a uma vídeo aula de Everson Borges Volcao que mostra como trabalhar com herança visual de formulários. http://www.devmedia.com.br/articles/visualizacomponente2.asp?comp=250p

*www.devmedia.com.br/articles/viewcomp.asp?comp=1716&hl=*heran%E7a**

Trabalhando com strings

No dia-a-dia do desenvolvimento, precisamos trabalhar com *strings* para, por exemplo, remover um caractere específico, remover acentos de um texto entre outros. Vamos então criar métodos para esses exemplos e podermos usar o código em toda a aplicação.

Começaremos com o exemplo de remover um caractere de um texto. Esse exemplo é utilizado quando precisamos remover um "-" ou qualquer outro caractere do texto. Vamos usar o código, onde passamos o caractere que queremos remover do texto (segundo parâmetro). O código retorna o texto sem a *string*, como podemos ver na **Listagem 4**.

No código, procuramos a posição do caractere no texto, usando o *Pos*. Com a posição, fizemos um laço e removemos o caractere do texto. Para testar, vamos adicionar dois *Edits* e um *Button* em um formulário. No primeiro vamos escrever o texto com o caractere que vamos remover.

No *Button* digite o seguinte código:

```
Edit2.Text := RemoveChar('-', Edit1.Text);
```

Listagem 4. Removendo um caractere do texto

```
function RemoveChar (const Ch: Char;
  const S: string): string;
var
  Posicao: integer;
begin
  Result := S;
  Posicao := Pos(Ch, Result);
  while Posicao > 0 do
  begin
    Delete(Result, Posicao, 1);
    Posicao := Pos(Ch, Result);
  end;
end;
```

Listagem 5. Removendo letras acentuadas

```
function RemoveAcento(Str: string): string;
const
  ComAcento = 'àâêôûãõáéíóúçüÀÂÊÔÛÃÕÁÉÍÓÚÇÜ';
  SemAcento = 'aaeouaoaeioucuAAEOUAOAEIOUCU';
var
  x: Integer;
begin
  for x := 1 to Length(Str) do
  if Pos(Str[x], ComAcento) <> 0 then
    Str[x] := SemAcento[Pos(Str[x], ComAcento)];
  Result := Str;
end;
```

Listagem 6. Gerador de senhas aleatórias

```
function GeraSenha (aQuant: integer): string;
var
  i: integer;
const
  str = '1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz';
begin
  for i:= 1 to aQuant do
  begin
    Randomize;
    Result := Result + str[Random(Length(str))+1];
  end;
end;
```

Listagem 7. Abrindo uma janela popup usando Java Script

```
procedure OpenWindow (aPage: System.Web.UI.Page;
  aRedirect, aWidth, aHeight: string);
var
  aScript: StringBuilder;
begin
  { abre uma nova janela do browser }
  aScript := StringBuilder.Create;
  aScript.Append('<script language="JavaScript">');
  aScript.Append('window.open("'+ aRedirect +
    '", "", "resizable=no, menubar=no, scrollbars=yes,
    status=yes, left=350, top=150, width='+ aWidth +
    ', height=' + aHeight + '")');
  aScript.Append('</script>');
  if not aPage.IsClientScriptBlockRegistered(
    'client') then
    aPage.RegisterClientScriptBlock('client',
      aScript.ToString);
end;
```

Com o código, vamos remover todas as "-" que estiverem no texto. Note que estamos apenas removendo e não substituindo. Veja na **Figura 1** o exemplo em execução.

Removendo letras acentuadas

Neste exemplo, vamos criar uma função que retorna o texto passado como parâmetro sem letras acentuadas. O código é o da **Listagem 5**.

No código temos as letras acentuadas da língua portuguesa, como também as mesmas sem acento, sendo preenchidas como constantes. Assim, fizemos um laço pegando o tamanho do texto como parâmetro e verificamos se existe a letra acentuada no texto e substituímos pela sem acento.

Veja na **Figura 2** o exemplo em execução, onde usamos a mesma técnica do exemplo anterior.

Gerando senhas aleatórias

Em alguns sistemas quando criamos o usuário, precisamos indicar uma senha aleatória gerada pelo sistema. Para isso, use o código da **Listagem 6**.

No código anterior, usamos *Random* e *Randomize* para pegar aleatoriamente a letra da constante (com letras maiúsculas, minúsculas e números) para criar

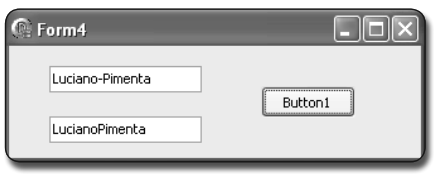


Figura 1. Removendo caractere de um texto

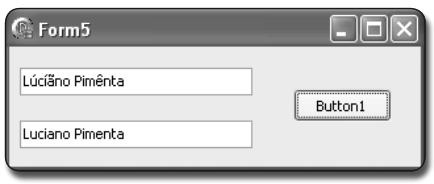


Figura 2. Substituindo letras acentuadas

Figura 3. Gerador de senhas

a senha. Passamos como parâmetro a quantidade de caracteres que a senha deve ter.

Se quiser, você pode adicionar caracteres especiais para que a senha fique mais segura. Veja na **Figura 3** o exemplo em execução.

Trabalhando com Java Script

Para *ASP.NET* também podemos criar nosso códigos personalizados. Os mais comuns são códigos para trabalhar com *JavaScript* para abrir formulários, executar código e mostrar mensagens em tela, mas também podemos desabilitar e limpar controles como na aplicação Win32.

Podemos abrir páginas *pop-up* utilizando *JavaScript*, assim reaproveitando o código em uma *unit*, não precisamos duplicar o código em cada página. O código para chamar uma página *ASPX* com *popup* está na **Listagem 7**.

No código, passamos como parâmetro o objeto *Page*, o nome da página que queremos abrir e o tamanho da janela. Você pode criar mais parâmetros para parametrizar ainda mais o método. Criamos uma variável do tipo *StringBuilder*, que é mais otimizada para concatenar *strings*.

Adicionamos na *StringBuilder* a declaração de uma função *JavaScript* (*tags*) chamando o *open* do objeto *window*. Passamos como parâmetros as opções existentes, que você pode conhecer em: *http://msdn.microsoft.com/en-us/library/ms536651(VS.85).aspx*.

Listagem 8. Executando scripts na Web

```
procedure ExecuteScript(aPage: System.Web.UI.Page;
  aScript: string);
var
  Script: StringBuilder;
begin
  Script := StringBuilder.Create;
  Script.Append('<script language="JavaScript">');
  Script.Append(aScript);
  Script.Append('</script>');
  if not aPage.IsClientScriptBlockRegistered(
    'client') then
    aPage.RegisterClientScriptBlock('client',
      Script.ToString);
end;
```

Listagem 9. Caixas de Mensagens com JavaScript

```
procedure MessageScript(aPage: System.Web.UI.Page;
  aTexto: string);
var
  Script: StringBuilder;
begin
  Script := StringBuilder.Create;
  Script.Append('<script language="JavaScript">');
  Script.Append('alert("'+ aTexto + '");');
  Script.Append('</script>');
  if not aPage.IsClientScriptBlockRegistered(
    'client') then
    aPage.RegisterClientScriptBlock('client',
      Script.ToString);
end;
```

Listagem 10. Habilitando/desabilitando controles ASP.NET

```
procedure EnableDisableControls (
  aControls: ControlCollection;
  aValue: Boolean);
var
  i, y: integer;
begin
  { habilita e desabilita controles }
  for i := 0 to aControls.Count - 1 do
  begin
    { controles de tela esta dentro de HTMLForm }

    if (aControls.Item[i] is HtmlForm) then
    begin
      for y := 0 to aControls.Item[i].Controls.Count - 1 do
        if (aControls.Item[i].Controls.Item[y] is WebControl) then
          (aControls.Item[i].Controls.Item[y] as WebControl).Enabled := aValue;
    end;
  end;
end;
```

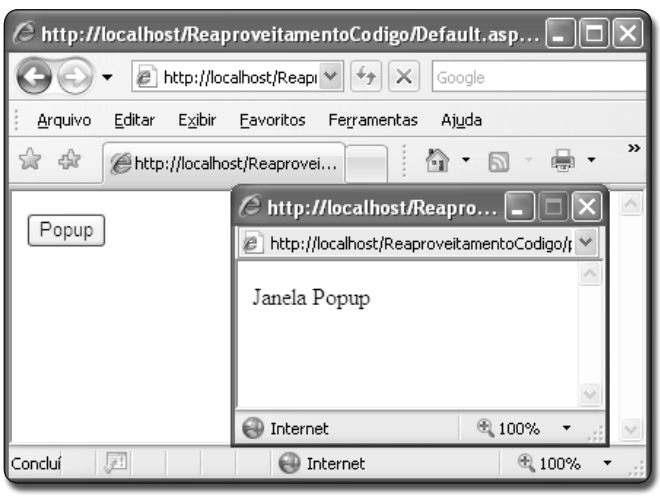


Figura 4. Abrindo janelas popup no ASP.NET

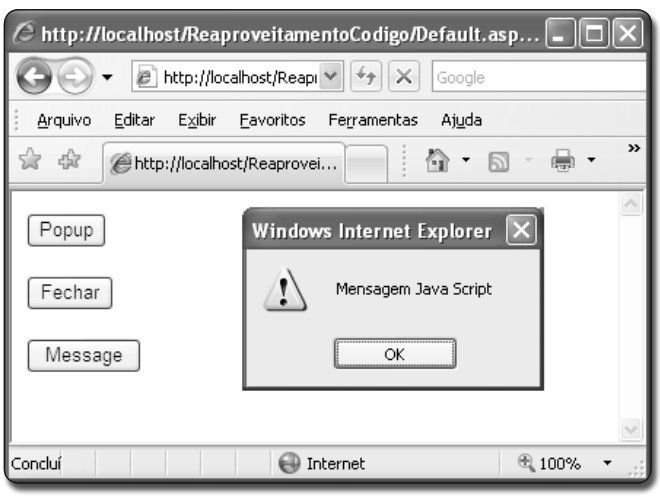


Figura 5. Caixas de mensagens no ASP.NET com JavaScript

Listagem 11. Limpando controles de tela no ASP.NET

```
procedure ClearControls (aControls: ControlCollection);
var
  i, y: integer;
begin
  { limpa controles de tela }
  for i := 0 to aControls.Count - 1 do
  begin
    if (aControls.Item[i] is HtmlForm) then
    begin
      for y := 0 to aControls.Item[i].Controls.Count - 1 do
      begin
        { controles mais comuns }
        if (aControls.Item[i].Controls.Item[y] is TextBox) then
          (aControls.Item[i].Controls.Item[y] as TextBox).Text := '';
        if (aControls.Item[i].Controls.Item[y] is RadioButtonList) then
          (aControls.Item[i].Controls.Item[y] as RadioButtonList).SelectedIndex := -1;
        if (aControls.Item[i].Controls.Item[y] is DropDownList) then
          (aControls.Item[i].Controls.Item[y] as DropDownList).SelectedIndex := -1;
        if (aControls.Item[i].Controls.Item[y] is ListBox) then
          (aControls.Item[i].Controls.Item[y] as ListBox).SelectedIndex := -1;
      end;
    end;
  end;
end;
```

Listagem 12. Criptografando textos

```
uses System.Web.Security;
...
function EncryptaSenha (aSenha: string): string;
begin
  Result := FormsAuthentication.HashPasswordForStoringInConfigFile(
  aSenha, 'MD5');
end;
```

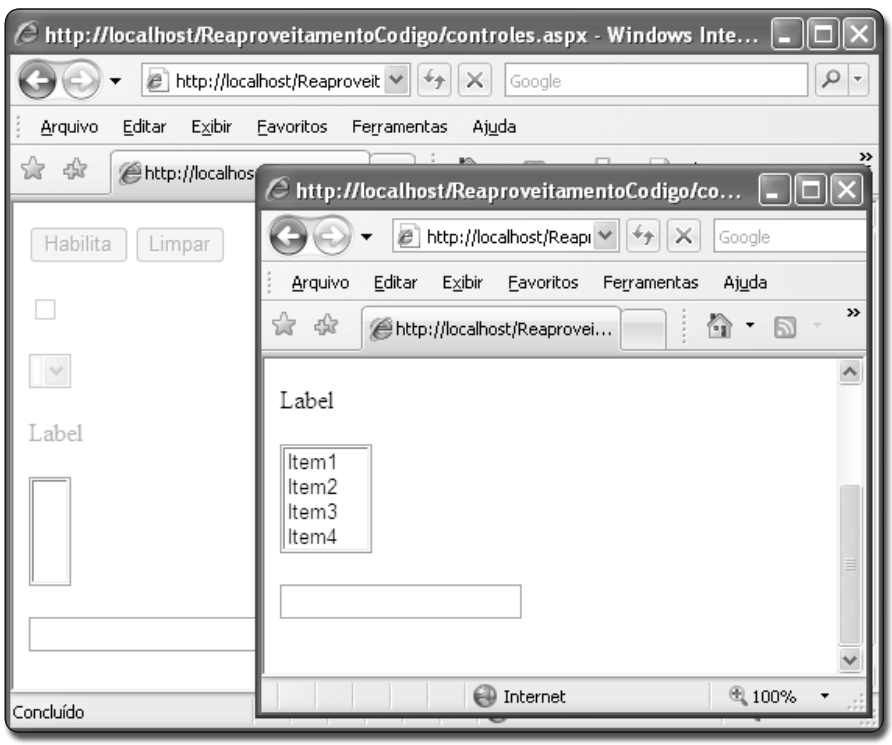
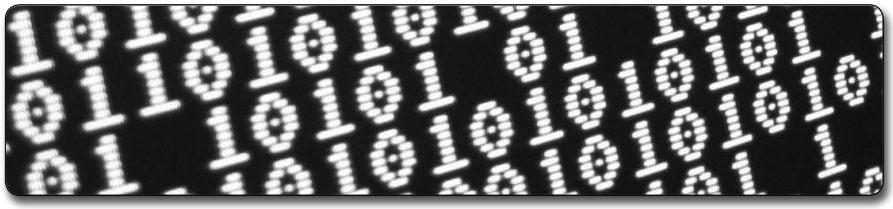


Figura 6. Trabalhando com controles de tela



Por fim, registramos o bloco de código chamando *RegisterClientScriptBlock* do objeto *Page*. Em uma aplicação *ASP.NET* use o código e teremos o exemplo da **Figura 4**.

Executando JavaScript

E se quisermos, por exemplo, executar determinado código *JavaScript* como fechar um formulário (browser)? Podemos criar um novo método, adaptado do exemplo anterior, usando o código da **Listagem 8**.

Veja que o código é bastante semelhante ao anterior, com a diferença que passamos o parâmetro *aScript* no meio da tag *<script>*. Para executar o código, precisamos apenas usar:

```
ExecuteScript(Page, 'window.close();');
```

Mensagens com JavaScript

Podemos usar o método anterior para usar o *alert* do *JavaScript*, mas também podemos criar um método específico para a caixa de mensagem. Veja o código da **Listagem 9**.

Nota do DevMan

O MD5 (Message-Digest algorithm 5) é um algoritmo de *hash* de 128 *bits* unidirecional desenvolvido pela *RSA Data Security, Inc.*, descrito na *RFC 1321*, e muito utilizado por *softwares* com *protocolo ponto-a-ponto* (*P2P*, ou *Peer-to-Peer*, em inglês), verificação de integridade e logins.

Foi desenvolvido em 1991 por Ronald Rivest para suceder ao MD4 que tinha alguns problemas de segurança. Por ser um algoritmo unidirecional, uma *hash* md5 não pode ser transformada novamente no texto que lhe deu origem. O método de verificação é, então, feito pela comparação das duas hash (uma da base de dados, e a outra da tentativa de login). O MD5 também é usado para verificar a integridade de um arquivo através, por exemplo, do programa *md5sum*, que cria a *hash* de um arquivo. Isso pode se tornar muito útil para *downloads* de arquivos grandes, para programas *P2P* que constroem arquivos através de porções e estão sujeitos à corrupção dos mesmos. Como autenticação de login é utilizada em vários sistemas operacionais unix e em muitos sites com autenticação.

Para testar, basta usar:

```
MessageScript(Page, 'Mensagem Java Script');
```

Veja o exemplo em execução na **Figura 5**.

Trabalhando com controles de tela

Podemos criar métodos para, por exemplo, habilitar/desabilitar e limpar controles de tela. Para habilitar/desabilitar use o código da **Listagem 10**.

Veja que precisamos percorrer a coleção de controles, onde passaremos a coleção de controles do objeto *Page*. Para poder verificar os *WebControls* precisamos achar o tipo *HtmlForm* e depois verificar os *WebControls*.

Para limpar controles de tela (*TextBoxs*, *RadioButtonList*, *DropDownList*, *ListBox* etc.), usamos um código bastante semelhante, conforme a **Listagem 11**.

Para usar os métodos, use o seguinte código:

```
{ limpar controles de telas }
ClearControls(Page.Controls);
{ habilitar/desabilitar controles }
EnableDisableControls(Page.Controls, false);
```

Veja na **Figura 6** o exemplo em execução.

Criptografia

Para criptografar textos no *ASP.NET* o framework possui métodos prontos para usar, portanto podemos usar esse método para nossa biblioteca. Para isso, basta usarmos o código da **Listagem 12**.

O *HashPasswordForStoringInConfigFile* retorna uma *string* criptografada com o formato passado como parâmetro (MD5), onde podemos ainda ter o formato SHA1. Uma característica interessante é que não podemos "descriptografar" a *string*.

O método é indicado para armazenar dados criptografados em arquivos de configuração, mas podemos usá-los para criptografar senhas por exemplo. Veja na **Figura 7** o texto criptografado.

Conclusão

Vimos neste artigo, como reaproveitar código *Win32* e *ASP.NET* usando uma biblioteca para que os métodos possam ser usados em toda a aplicação ou até em várias aplicações. Alguns exemplos *Win32* foram desenvolvidos por mim e outros encontrados na internet.

Os exemplos não continham a indicação do autor, por isso não pudemos mencionar o mesmo nos exemplos. Os exemplos em *ASP.NET* foram desenvolvidos por mim e podem ser usados em suas aplicações Web.

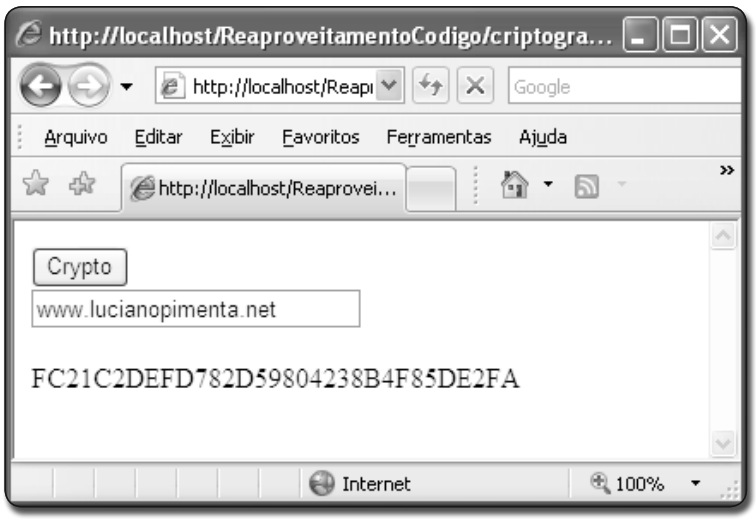
Um grande abraço a todos e sucesso em seus projetos! ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback



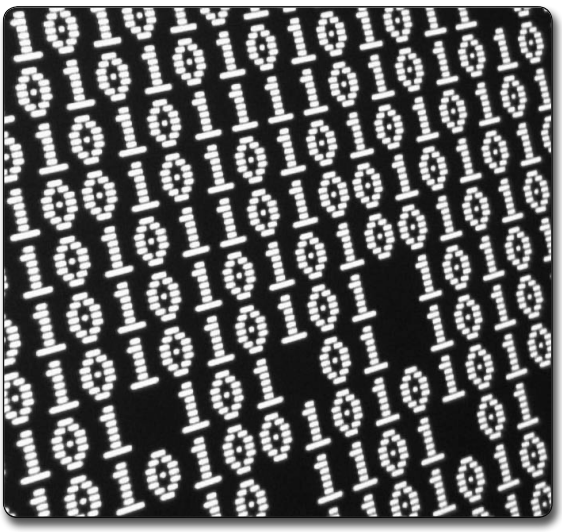
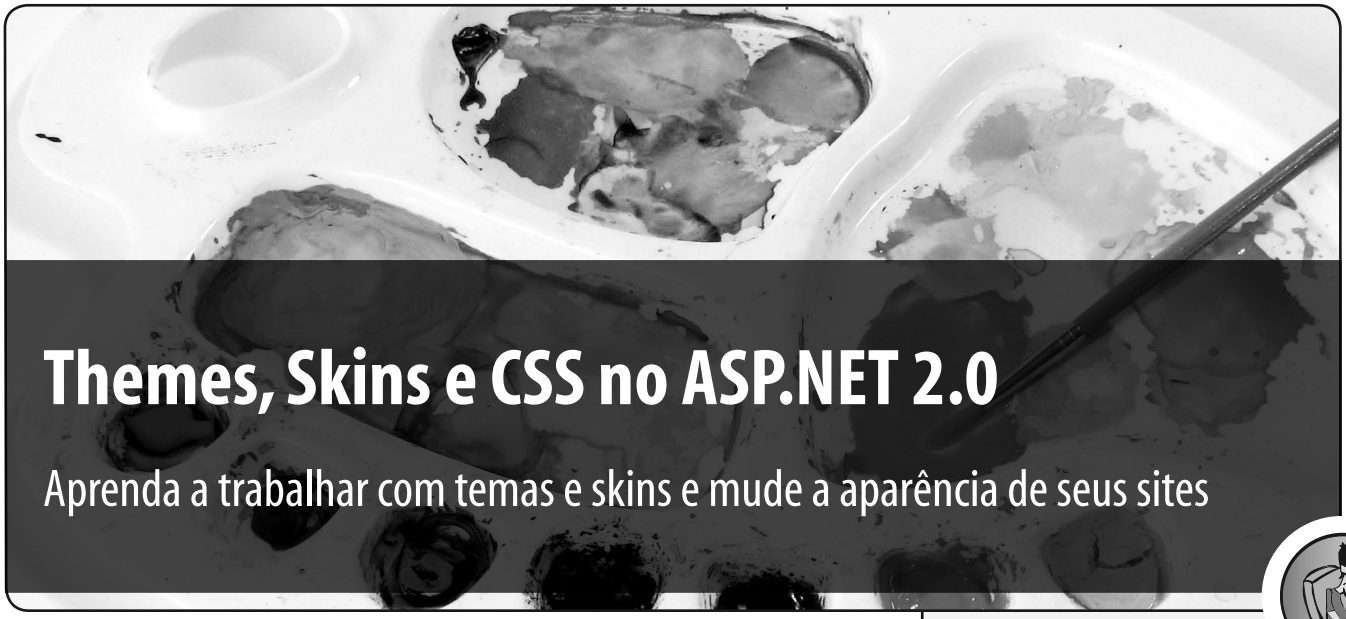


Figura 7. Criptografando textos com o formato MD5

Nesta seção você encontra artigos para iniciantes na linguagem Delphi



Themes, Skins e CSS no ASP.NET 2.0

Aprenda a trabalhar com temas e skins e mude a aparência de seus sites

Guinther Pauli
(guinther@devmedia.com.br)
É Bacharel em Sistemas de Informação, Microsoft Certified: MCP, MCAD, MCSD.NET, Borland Certified: Delphi 6, 7, 2005, 2006, Web e Kylix. Editor Geral das Revistas .net Magazine, ClubeDelphi. Trabalha com ASP.NET há 8 anos.

Adriano Santos
(falecom@adrianosantos.pro.br)
é desenvolvedor Delphi desde 1998. Professor e programador PHP. Bacharel em Comunicação Social pela Universidade Cruzeiro do Sul, SP. É Editor Técnico, Colunista e Membro da Comissão Editorial das revistas ClubeDelphi. Gerente de Desenvolvimento da SoftPark, empresa parceira Borland. Mantém o blog Delphi to Delphi (www.delphitodelphi.blogspot.com) com dicas, informações e tudo sobre desenvolvimento Delphi

Sabe aqueles aplicativos em que você pode escolher o layout gráfico da interface? O aplicativo possui vários *skins* que podem ser trocados a hora que desejarmos, assim uma hora o aplicativo é amarelo outra azul, dependendo do *skin* definido? Conhece isso? O próprio *Windows* permite esse tipo de configuração quando trocamos o esquema de cores nas propriedades do vídeo.

E o que você acha de embutir essa funcionalidade em seus projetos Web? Pois é isso que veremos neste artigo. O *ASP.NET 2.0* trouxe os *Themes* e *Skins* que permitem criar múltiplas formatações de *layout*, de uma forma bem simples e intuitiva.

A pasta App_Themes

Para mostrarmos como fazer para que nossas aplicações possam ter aparências diferentes, vamos começar criando um novo *Web Site ASP.NET* usando o RAD Studio 2007 e para isso basta acionar o menu *File>New>ASP.NET Web Applica-*

Resumo DevMan

A interface de um site, sem dúvida, é uma das partes mais importantes dele, pois valoriza o Web Site. Poder trocar a "cara" do site com apenas alguns cliques, é simplesmente fantástico, porque deixa de ser um problema na vida do desenvolver para ser um prazer.

Veremos como é fácil criarmos diversos temas para nossos Web Sites utilizando-se de recursos do RAD Studio 2007 com .net 2.0, assim como é facílimo trocar de tema no sistema operacional Windows.

Nesse artigo veremos

- Utilização de temas e skins em aplicações Web;
- Como criar um arquivo CSS para cada Tema;
- Aplicando tema e estilo ao mesmo tempo.

Qual a finalidade

- Com temas e skins, fica muito mais fácil montar uma página atraente e elegante.

Quais situações utilizam esses recursos?

- Os temas e os skins valorizam a página, são importantes para manter padrões de página em um mesmo projeto.

tion – Delphi for .NET. Salve a aplicação com o nome *SkinsThemes.dproj* em um diretório de sua escolha. Adicione na página um *Label*, um *TextBox* e um *Button*. Vamos formatá-los dando destaque para a cor vermelha (**Figura 1**).

Observe que fizemos apenas simples formatações, basicamente alteramos as propriedades *BackColor*, *ForeColor* e *Size* dos controles. Até aqui tudo bem, não vimos novidade nenhuma. Vamos criar uma estrutura de pastas para utilizarmos os *Themes* e *Skins*. Clique com o botão direito do mouse no nome do projeto, ou seja, em *ThemesSkins.dll* no *Project Explorer* e selecione *Add>New>Folder*. Dê o nome de *App_Themes* a essa pasta. Agora dê um clique de direita nessa pasta e crie outra, dessa vez o nome *Vermelho*. Repita esse processo e crie duas novas pastas e nomeie-as para *Azul* e *Verde* como mostra a **Figura 2**.

Veja que *App_Themes* é um nome especial de pasta para o nosso projeto, assim como *App_Data*, portanto precisa estar exatamente com esse nome. Como você já deve estar imaginando, cada pasta corresponde a um *Theme* no projeto.

Arquivos skin

Nessa estrutura, cada pasta conterá um tema para a aplicação. Um tema pode ser composto de vários componentes, necessários para definir uma interface. O principal componente de um *Theme* é o *Skin*.

No *ASP.NET 2.0*, um *Skin* é um arquivo onde, como o próprio nome sugere, define a "pele" de uma página *ASPX*. A melhor forma de entender o *Skin* é na prática, portanto vamos a ela. Clique com o botão direito sobre a pasta *Vermelho* e escolha a opção *Add>New>Other* e em *Delphi for .NET Projects>New ASP.NET Files* escolha o template *Skin File*. O arquivo é criado com o nome *SkinFile.skin*, clique nele e renomeie para *ControlesComuns.skin*. Repita esses passos para cada uma das pastas e dê sempre o mesmo nome para o arquivo de *Skin*.

Veja que o arquivo *Skin* foi criado e nele temos apenas um comentário com algumas instruções sobre os *Skins*. Outra característica que você pode observar é que um arquivo *Skin* é como um arquivo ASPX, porém não tem um modo de *design*.

Listagem 1. Conteúdo do arquivo ControlesComuns.skin para o tema Azul

```
<asp:Label runat="server" Font-Bold="True"
  Font-Size="X-Large" ForeColor="Red" Text="Label"/>
<asp:TextBox runat="server" BackColor="Red"
  Font-Bold="True" Font-Size="Medium" ForeColor="White"/>
<asp:Button runat="server" Font-Bold="True"
  ForeColor="Red" Text="Button" />
```

Listagem 2. Conteúdo do arquivo ControlesComuns.skin para o Theme Azul

```
<asp:Label runat="server" Font-Bold="Blue"
  Font-Size="X-Large" ForeColor="Red" Text="Label"/>
<asp:TextBox runat="server" BackColor="Blue"
  Font-Bold="True" Font-Size="Medium" ForeColor="White"/>
<asp:Button runat="server" Font-Bold="True"
  ForeColor="Blue" Text="Button" />
```

Mas não há porque se desesperar, sempre encontramos um jeitinho para facilitar o trabalho. Primeiramente, podemos apagar todo o conteúdo do arquivo *Skin*. Em seguida, vamos voltar ao *Default.aspx* e ir até o *Source* da página. Marque e copie todo o conteúdo do código da *Default.aspx* e cole em nosso arquivo *ControlesComuns.skin*.

E como mostra a **Listagem 1**, mantenha nesse arquivo apenas as declarações dos controles *Label*, *TextBox* e *Button*, apagando todo o resto do código. Em seguida apague a propriedade ID e *Text* de cada um dos controles. Seu arquivo deve ficar exatamente igual ao demonstrado na **Listagem 1**.

Para completar o primeiro exemplo, crie, se ainda não o fez, os arquivos de *Skin* para as demais pastas. Ambos devem ter o mesmo nome *ControlesComuns.skin*, como mencionado anteriormente, e também o mesmo conteúdo. Para finalizar, no *Skin* do *Theme Vermelho*, substitua todas as ocorrências da palavra *Red* por *Blue*, como mostra a **Listagem 2**. Repita esse processo com a pasta *Verde* e refaça as cores colocando *Green* nos locais necessários.

Aplicando Themes e Skins em uma página aspx

Com as pastas de *Themes* e os arquivos *Skins* criados, vamos aplicá-los em uma página *ASPX*. Volte à *Default.aspx*, e para termos uma melhor visão, retire as formatações feitas nos controles, ou apague os controles existentes na página e os inclua novamente. A referência ao tema terá que ser adicionada manualmente no arquivo *Web.config*, isso mesmo! Teremos que abrir o arquivo e incluir uma

Figura 1. Formulário formatado com destaque na cor vermelha

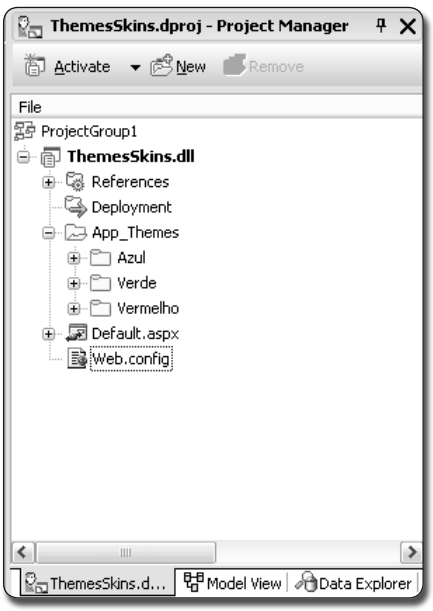


Figura 2. Pastas de temas no RAD Studio

linha na seção *system.web*. Vejamos como fazer isso. Apenas clique duas vezes no arquivo mencionado para que possamos ver seu código-fonte. Localize a seção *<system.web>* e logo abaixo dela inclua a linha a seguir:

```
<pages theme="Azul" />
```

Perceba que estamos definindo que nosso Web Site está sendo vinculado ao tema *Azul*. Poderíamos ter criado a pasta com outro nome, como *Tema1*, *Tema2* ou qualquer outra coisa. Então bastaria associar o nome criado à propriedade *theme* na *tag pages*, como vimos anteriormente. Experimente agora salvar a página e executá-la. Verá nitidamente que funciona

Listagem 3. Inclusão de outros TextBoxes

```
<asp:TextBox runat="server" BackColor="Blue"
  Font-Bold="True" Font-Size="Medium" ForeColor="White"/>
<asp:TextBox runat="server" BackColor="Blue"
  Font-Bold="True" Font-Size="Medium" ForeColor="White"/>
<asp:TextBox runat="server" BackColor="Blue"
  Font-Bold="True" Font-Size="Medium" ForeColor="White"/>
```

Listagem 4. Definindo SkinID

```
<asp:TextBox runat="server" BackColor="Blue"
  Font-Bold="True" Font-Size="Medium" ForeColor="White"/>
<asp:TextBox runat="server" BackColor="DarkBlue" SkinId="DarkBlue"
  Font-Bold="True" Font-Size="Medium" ForeColor="White"/>
<asp:TextBox runat="server" BackColor="LightBlue" SkinId="LightBlue"
  Font-Bold="True" Font-Size="Medium" ForeColor="White"/>
```

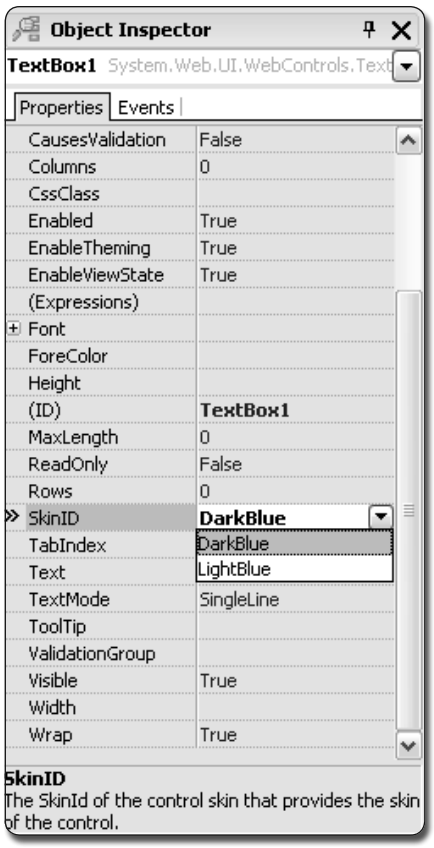


Figura 3. Definindo o SkinId do controle TextBox

Figura 4. TextBox definido com um Skin LightBlue

perfeitamente a troca de cores. Retorne ao RAD Studio e modifique o *Web.config* e refaça o teste.

Propriedade SkinID

Como podemos observar, as formatações definidas no arquivo *Skin* de cada um dos *Themes* foram automaticamente aplicadas nos controles, sem precisar realizar nenhuma configuração. Isso ocorreu, pois na definição dos controles que foram feitas no arquivo *Skin*, não definimos a propriedade *SkinID*.

Vamos abrir o arquivo *ControlesComuns.skin* do *Theme Azul*. Marque a linha de declaração do *TextBox*, e copie. Logo abaixo, cole a linha copiada, incluindo mais dois *TextBoxes* no arquivo. Como mostra a **Listagem 3**, na primeira linha incluída vamos trocar o *BackColor* para *DarkBlue*, e inclua a propriedade *SkinId* que deverá ser igualada à *DarkBlue* também.

Faça o mesmo para a outra linha incluída, trocando o *BackColor* para *LightBlue* e incluindo a propriedade *SkinId*, também definida com *LightBlue*. Seu arquivo *Skin* deverá ficar idêntico ao da **Listagem 4**.

Volte à página *Default.aspx*, e veja que nosso controle continua definido como Azul. Isso porque temos uma definição do *TextBox* sem a propriedade *SkinId*. Essa é a formatação padrão. Clique no controle *TextBox* e na janela de propriedades, veja que temos uma propriedade chamada *SkinId*. Clicando nela, podemos escolher entre os formatos *DarkBlue* e *LightBlue* conforme **Figura 3**). A mudança não é instantânea, portanto selecione a opção que te agrada e execute a página para ver o resultado (**Figuras 3 e 4**).

Nota do DevMan

Cascading Style Sheets, ou simplesmente CSS, é uma linguagem de estilo utilizada para definir a apresentação de documentos escritos em uma linguagem de marcação, como HTML ou XML. Seu principal benefício é prover a separação entre o formato e o conteúdo de um documento.

Ao invés de colocar a formatação dentro do documento, o desenvolvedor cria um link (ligação) para uma página que contém os estilos, procedendo de forma idêntica para todas as páginas de um portal. Quando quiser alterar a aparência do portal basta, portanto, modificar apenas um arquivo.

Com a variação de atualizações dos navegadores (browsers) como Internet Explorer que ficou sem nova versão de 2001 a 2006, o suporte ao CSS pode variar. O Internet Explorer 6, por exemplo, tem suporte total a CSS1 e praticamente nulo a CSS2. Navegadores mais modernos como Opera, Internet Explorer 7 e Mozilla Firefox têm suporte maior, inclusive até a CSS 3, ainda em desenvolvimento.

A interpretação dos browsers pode ser avaliada com o teste Acid2, que se tornou uma forma base de revelar quão eficiente é o suporte de CSS, fazendo com que a nova versão em desenvolvimento do Firefox seja totalmente compatível a ele assim como o Opera já é.

Eis um exemplo de CSS:

```
body
{
  font-family: Arial, Verdana, sans-serif;
  background-color: #FFF;
  margin: 5px 10px;
}
```

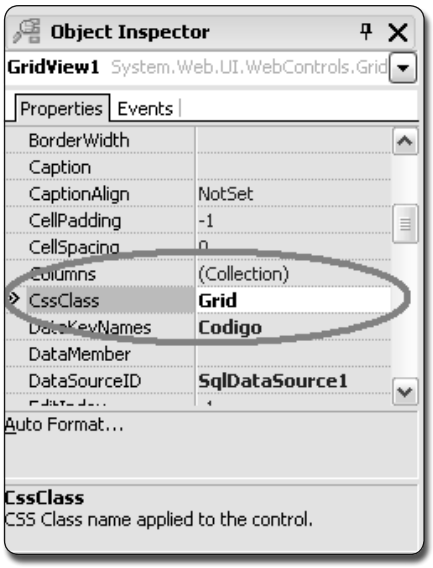


Figura 5. Definindo a propriedade CssClass

Cascade Style Sheet (CSS)

Outra possibilidade que temos com os *Themes* e *Skins* é o uso de arquivos *Cascade Style Sheet*, os famosos *CSS*. Podemos utilizá-los junto aos *Themes* de uma maneira bem simples. Faça o seguinte: clique com o botão direito sobre a pasta *Azul* que temos em *App_Themes* e escolha a opção *Add>New>Other>Web Documents>CSS StyleSheet* e dê o nome de *Azul.css*. Codifique o arquivo como na **Listagem 5**.

Observe que temos apenas uma classe em nosso arquivo *CSS*, chamada *grid*. Nela, temos as formatações necessárias para um *GridView*, onde estamos dando um destaque para a cor azul clara. Crie também os arquivos *CSS* correspondentes para a pasta *Verde* e *Vermelho*, como mostra a **Listagens 6**.

Vamos voltar à página *Default.aspx* para testar o uso dos arquivos CSS com os *Themes*. Inclua um *GridView* à página. Nas propriedades do controle, informe *grid* na propriedade *CssClass*, como mostra a **Figura 5**.

Novamente, a alteração ocorre diretamente na IDE então para notar a diferença, salve o projeto e execute-o. Troque o tema no *Web.config* para *Vermelho* e *Verde* e teste novamente a aplicação (**Figura 6**).

Listagem 5. Arquivo Azul.css

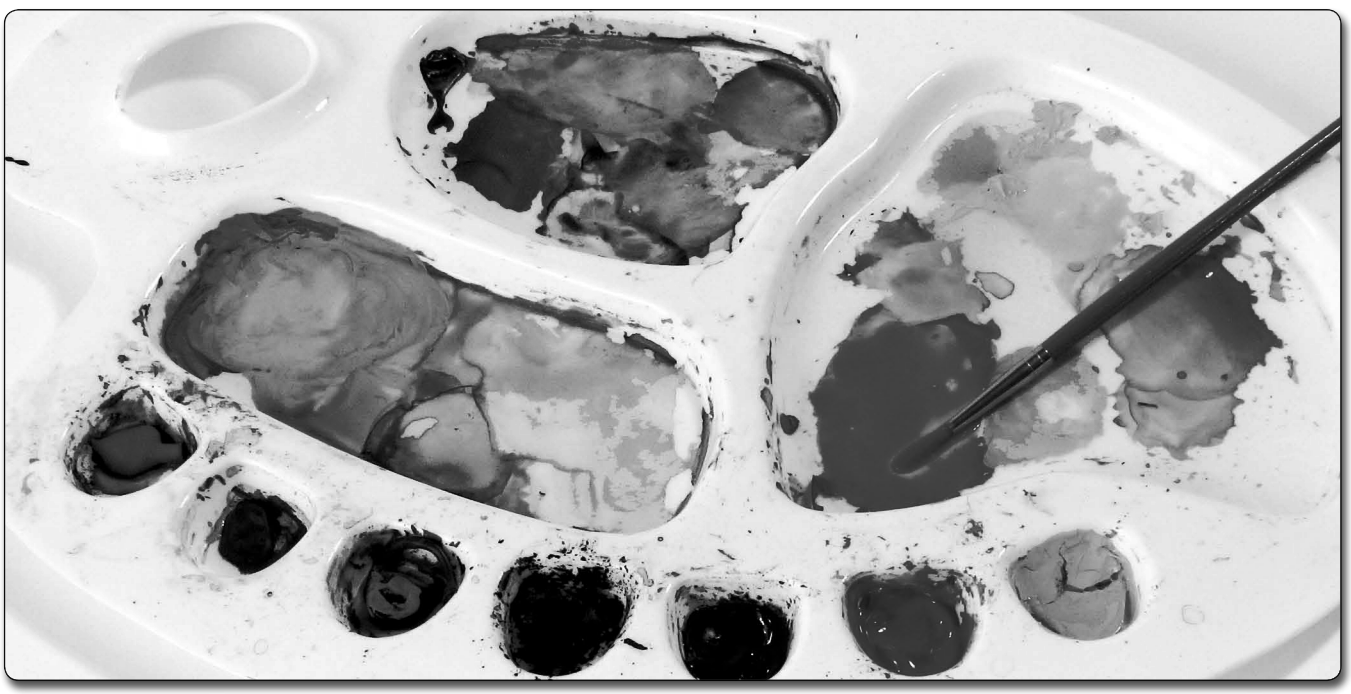
```
.grid{
    border-bottom : 1PX SOLID #000000;
    border-top : 1PX SOLID #000000;
    backgroup-color: lightblue;
    padding-left: 5px;
    padding-right: 2px;
    font-family: Verdana, sans-serif;
    font-size: 11px;
    font-weight: normal;
    color: #4C4A4B;
}
```

Listagem 6. Arquivos CSS para Verde e Vermelho

```
//Verde
.grid{
    border-bottom: 1PX SOLID #000000;
    border-top: 1PX SOLID #000000;
    background-color: lightgreen;
    padding-left: 5px;
    padding-right: 2px;
    font-family: Verdana, sans-serif;
    font-size: 11px;
    font-weight: normal;
    color: #4C4A4B;
}
//Vermelho
.grid{
    border-bottom : 1PX SOLID #000000;
    border-top : 1PX SOLID #000000;

    background-color: Orange;

    padding-left: 5px;

    padding-right: 2px;
    font-family: Verdana, sans-serif;
    font-size: 11px;
    font-weight: normal;
    color: #4C4A4B;
}
```

Listagem 7. Código do GridView

```
<asp:GridView runat="server" cssclass="Grid"
  autogeneratecolumns="False" datakeynames="Codigo" forecolor="Black">
  <HeaderStyle backcolor="Green" forecolor="White">
  </HeaderStyle>
</asp:GridView>
```



GridView

Com esse exemplo que acabamos de realizar, vimos que é possível ter um arquivo *CSS* para cada *Theme*. Agora, vamos fazer um teste mais avançado com *Themes* e *Skins*, incluindo as formatações do *GridView* nos arquivos de *Skin*. Na página *Default.aspx* vamos apenas realizar uma formatação no cabeçalho, alterando a propriedade *BackColor* para *Green*, e a propriedade *ForeColor* para *White* (**Figura 7**).

Feito isso, vamos copiar o código do *GridView* para o arquivo *Skin* do *Theme Verde*. Retirando a propriedade *ID*, veja na **Listagem 7** o código do *GridView* que deve ser copiado ao arquivo *ControlesComuns.skin* da pasta *Verde*.

Copie esse mesmo trecho de código para os arquivos correspondentes dos *Themes Azul* e *Vermelho*, trocando a propriedade *BackColor* para *Blue* e *Red* respectivamente. E assim, podemos conferir o resultado apenas trocando as propriedades *theme* no arquivo *Web.config* da página (não esquecendo de trocar as propriedades *BackColor* e *ForeColor* do *GridView* para o valor padrão), como mostra a **Figura 8** no caso do *Theme Azul*.

Se preferir, você pode excluir o *GridView* e incluir um novo na página. Você verá que as formatações são prontamente aplicadas.

Conclusão

Como você deve ter observado, esse foi um exemplo bem simples e introdutório. Como já foi dito, um *Theme* não é composto por apenas um arquivo *Skin*. Podemos definir vários arquivos *Skins* em uma pasta de *Theme*, podendo até criar um arquivo por controle.

Para facilitar a criação dos arquivos *Skins*, devemos sempre criar as interfaces em uma página *ASPX* real, e depois copiar o código-fonte. Isso facilita muito o trabalho com controles mais complexos, como foi o caso do *GridView*.

Também vimos que se você utilizar arquivos *CSS* em seus projetos poderá facilmente adaptá-los aos *Themes* e *Skins*, inclusive pode ter um arquivo *CSS* por *Theme*. Resumindo, a utilização dos *Themes* e *Skins* é uma ótima forma de criar várias interfaces em nossa aplicação Web, e facilmente aplicar essas interfaces em nossas páginas. ●

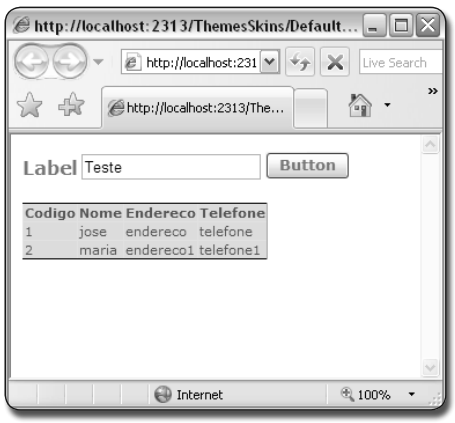


Figura 6. Exemplo do uso de CSS com Theme

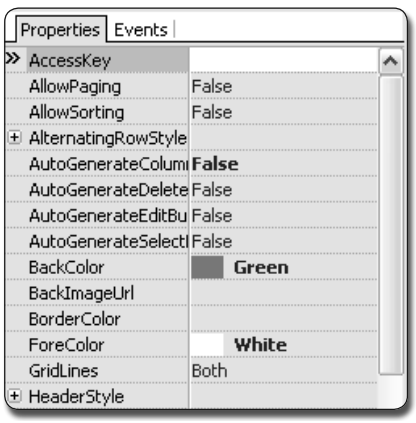


Figura 7. Propriedades do HeaderStyle do GridView

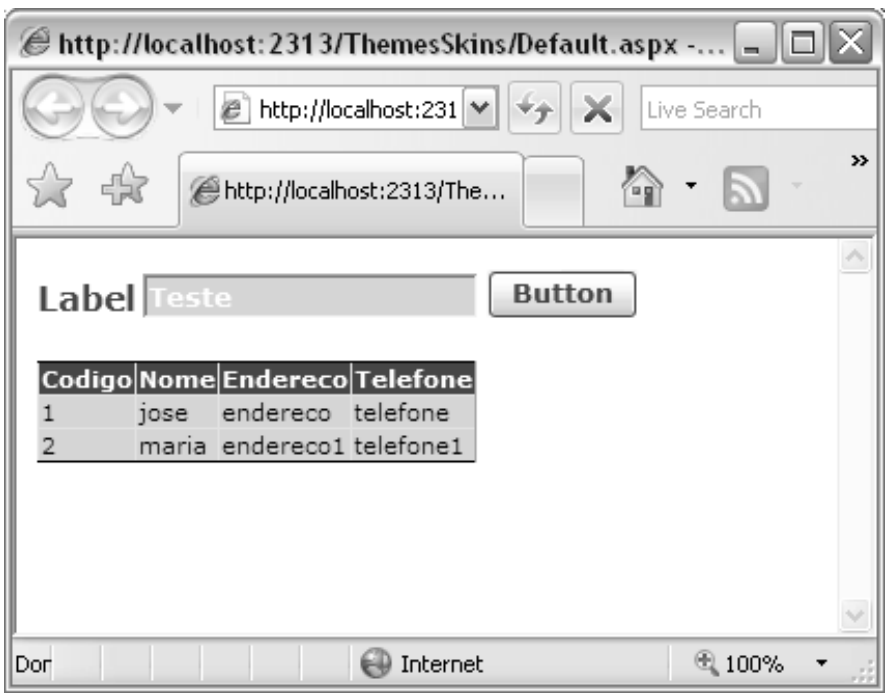


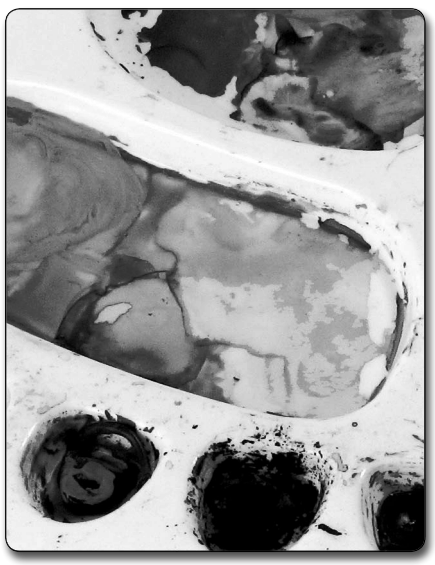
Figura 8. Configurando o GridView no arquivo Skin do Theme

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback



VENHA FAZER PARTE DA MELHOR EQUIPE DE HOSTING DO BRASIL

MAIS DE 100 VAGAS PARA RJ E SP

Acesse o site **www.alog.com.br** e cadastre seu currículo no link **carreira**

- SUPORTE TÉCNICO
- MONITORAMENTO
- GERÊNCIA DE SERVIÇOS
- OPERAÇÕES
- REDES
- INFRA-ESTRUTURA

ALOG DATA CENTERS DO BRASIL
Líder em Hosting Gerenciado

- Dois datacenters próprios - RJ e SP
- 99% de renovação contratual
- 850 clientes corporativos
- 8 mil m² de área construída
- Melhor empresa de internet pelo INFO 200 de 2008
- Destaque do ano no segmento Serviços de Internet do Anuário Telecom 2007

www.alog.com.br I 0800 282 3330

ALOG
DATA CENTERS DO BRASIL

Hosting Gerenciado® I Colocation I Contingência I Email Corporativo I Conectividade I Serviços Profissionais

Nesta seção você encontra artigos sobre a linguagem PHP e a ferramenta Delphi for PHP

Manipulação direta de dados

Aprenda comandos para manipulação de dados em SGBD com o PHP

Resumo DevMan

O mercado tecnológico do ponto de vista de armazenamento de dados está cada vez mais avançado. Há uma variedade enorme de bancos de dados disponíveis no mercado e com isso aumentam as chances de termos que portar nossas aplicações para cada um deles. A premissa básica hoje para qualquer profissional de desenvolvimento, é conhecer diversos bancos de dados e suas principais diferenças, sejam elas técnicas ou do ponto de vista de recursos, ou seja, conhecer o SGBD em que se vai trabalhar é extremamente necessário nos dias de hoje.

Veremos nesse artigo como conectar-se a dois bancos de dados bem conhecidos. Conheceremos as principais diferenças em termos de programação entre cada um. Faremos com que nossa aplicação se adapte a cada um.

Nesse artigo veremos

• Comandos necessários para acesso e manipulação de dados em SGBD's;

• Técnicas de adaptação de sistema a mais de um banco de dados;

Qual a finalidade

• Veremos nesse artigo algumas das principais técnicas de acesso a dados em PHP;

Quais situações utilizam esses recursos?

• A principal situação em que podemos aplicar essas técnicas consiste em poder compatibilizar nossos sistemas a mais de um banco de dados, poupando tempo na hora de adequação do mesmo;

Fabrício Desbessel

(fabricio@fabricio.pro.br)

é professor de Linguagem de Programação do Curso Técnico em Informática do Centro Tecnológico Frederico Jorge Logemann de Horizontina/RS e da FAHOR Faculdade Horizontina. Delphiano de coração está sempre disposto a provar que com o Delphi sempre teremos a melhor solução, até mesmo como PHP. Site www.fabricio.pro.br.

O *PHP* surgiu para simplificar ao máximo o desenvolvimento de aplicações WEB. Isso é claramente perceptível quando é necessário escrever um código para, por exemplo, buscar informações em um banco de dados. Com três comandos básicos acessamos o banco, fazemos a consulta e preparamos os dados para exibição.

Nesse artigo vamos aprender esses comandos utilizando dois Sistemas Gerenciadores de Bancos de Dados (SGBD): *MySQL* e *Firebird*. Vamos criar uma base de dados com uma tabela de clientes nos dois bancos de dados e manipular nossa conexão através de comandos e macetes disponíveis no *PHP*.

Ao final desse artigo você estará apto

a fazer consultas, inserções, alterações e exclusões no banco de dados escrevendo os comandos necessários, sem o auxílio de geradores de aplicação ou ferramentas de *webdesign*.

Criando o banco de dados no MySQL

Para a criação do banco de dados você deve ter instalado o *MySQL Server* (baixe em *dev.mysql.com*) e o *MySQL Administrador* (*dev.mysql.com/downloads/gui-tools/index.html*).

Para criar o banco de dados abra o *MySQL Administrator*, e faça o *login* em *localhost*. Clique em *Catalogs* e em *Schema* clique com o botão direito e escolha *Create New Schema* (**Figura 1**). Você também pode utilizar o atalho das teclas *CTRL+N*. No diálogo *Create new Schema*, informe *direto* para o campo *Schema name*. Após a criação do banco de dados devemos executar o *script* SQL para criação das tabelas e chaves. Para tanto, acesse o menu *Tools>MySQL Command Line Client*. Digite: "use direto;" e pressione ENTER.

Copie o código da **Listagem 1** e digite-o no *command*. Finalize pressionando ENTER novamente, para que as tabelas sejam criadas.

Criando o banco de dados no Firebird

Para a criação do banco de dados você deve ter instalado o *Firebird Server* (baixe em *http://www.firebirdsql.org/*) e, também, algum software para gerenciamento, como o gratuito *IBOConsole* que pode ser encontrado em *www.mengoni.it/Downloads/*. O interessante é que o *IBOConsole* tem tradução para a Língua Portuguesa. Basta acessar o menu *Console > Options* e escolher a opção *Language*. Os passos abaixo utilizarão o *IBOConsole* mas você pode utilizar outro gerenciador a sua escolha.

A primeira coisa a se fazer com o *IBOConsole* é conectar no banco de dados local. Se você acabou de instalar o *Firebird* e o *IBOConsole* é necessário acessar o menu *Servidor > Registrar*. Na tela exibida na **Figura 2** marque a opção *Local Server* e preencha o campo *Descrição*. Depois basta um clique duplo para fazer o login no banco. O nome de super-usuário no *Firebird* é *SYSDBA* e a senha é *masterkey*.

Nota: O usuário *SYSDBA* e a senha *masterkey* é um padrão nas instalações dos SGBD's *Firebird* e *Interbase*. É altamente recomendável a troca dessa senha, o que pode ser realizado no *IBOConsole* pelo menu *Servidor > Segurança de Usuários*.

Nota: Caso ocorra o erro *Client library fbclient.dll not found in the path. Please install it to use this functionality* ao tentar registrar o servidor no *IBOConsole*, basta copiar o arquivo *fbclient.dll* do diretório *bin* na pasta de instalação do *Firebird* diretamente para a pasta *C:\Windows\System32* e tentar novamente.

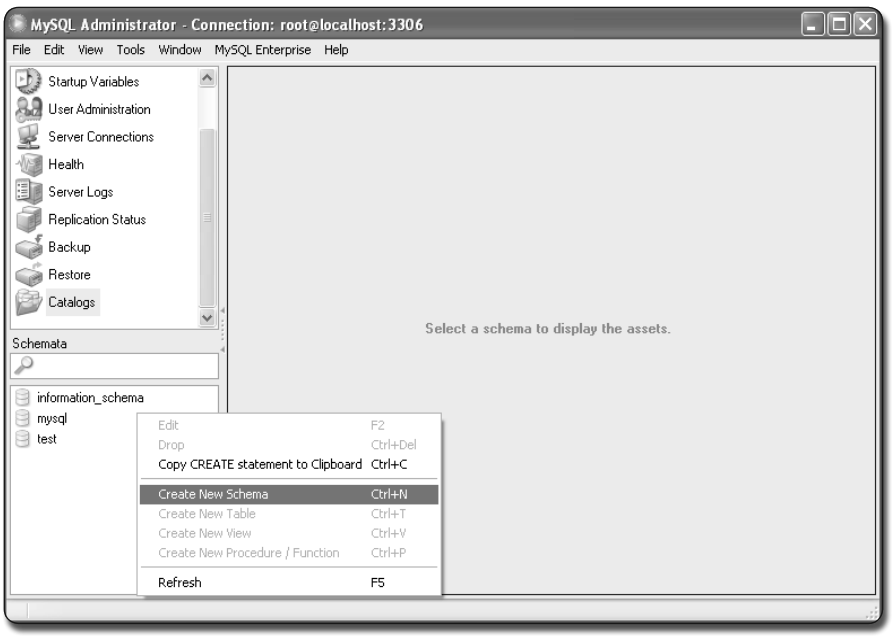


Figura 1. Criação do novo banco no MySQL Administrator

Listagem 1. Script para criação das tabelas no MySQL

```
CREATE TABLE 'clientes' (
  'id' int(10) unsigned NOT NULL auto_increment,
  'nome' varchar(45) NOT NULL,
  'datanasc' date NOT NULL,
  'fone' varchar(10) default NULL,
  'email' varchar(100) default NULL,
  PRIMARY KEY  ('id')
);
```

ClubeDelphi PLUS www.devmedia.com.br/clubedelphi/portal.asp
Acesse agora mesmo o portal do assinante ClubeDelphi e assista as aulas de Fabrício Desbessel que ensinam a instalar o banco de dados MySQL Server 5.0 e as ferramentas de administração.
www.devmedia.com.br/articles/viewcomp.asp?comp=4985

Figura 2. Registrando um banco de dados Firebird instalado localmente

Agora passamos para criação do banco de dados efetivamente. Acesse o menu *Banco de Dados > Criar Banco de Dados*. A primeira definição a ser realizada é do *File Name* onde, deve-se preencher com o caminho e nome do arquivo. Coloque, por exemplo: *C:\Direto.FDB*.

Depois no campo *Size (Pages)* digite o valor 4096 para definir o tamanho de paginação. No campo *SQL Dialect* certifique-se que está o valor 3 (três) pois , nesse dialeto é possível utilizar campos do tipo *date*. Preencha o campo *Alias* e pressione OK (**Figura 3**).

Após esse procedimento o banco de dados já estará criado e aberto, faltando ainda a criação da tabela de clientes. Clique no botão *SQL Interativo* (símbolo SQL) e na aba SQL digite o código contido na **Listagem 2**.

A **Listagem 2** tem o mesmo objetivo da **Listagem 1**, porém, no *Firebird*, precisamos de mais artifícios para criar um campo que seja auto incremento. É necessário criar um *Generator* que funciona como uma variável que conterá o último código utilizado. Para efetuar o incremento utiliza-se uma *Trigger* (gatilho) que é disparado quando houve uma inserção de registro na tabela e tem a função de preencher o campo com o novo valor obtido pela função *GEN_ID* que nada mais é que buscar o valor do *generator* atual e somar mais um.

Conectando ao banco de Dados

Para efetuar uma conexão ao banco de dados precisamos somente de um comando o *connect*. Se for conectar no banco de dados *MySQL* colocamos *mysql_connect*. Se for no Firebird utilizamos o *ibase_connect*. Agora vamos ver a sintaxe completa do *connect*:

Mysql:

```
mysql_connect(SERVIDOR, USUARIO, SENHA);
```

Exemplo: mysql_connect('localhost', 'root', 'root');

Firebird:

```
ibase_connect(SERVIDOR E PATH, USUARIO, SENHA);
```

Exemplo: ibase_connect('localhost:C:\Direto.FDB','SYSDBA','masterkey');

Tanto *mysql_connect* como *ibase_connect* retornam um link com a conexão se a mesma for bem sucedida ou *FALSE* caso ocorra um erro.

Uma diferença perceptível nos comandos é a necessidade de especificar o banco de dados para acesso no *Firebird*, durante a conexão. No *MySQL* essa especificação é realizada somente no momento de executar uma SQL, com um comando específico, explicado posteriormente.

No código da **Listagem 3** tem-se um pequeno script que tenta fazer a conexão com os bancos de dados criados. Para digitar esse código você pode utilizar um editor de texto simples. Eu particularmente digitei no *Delphi For PHP*, pois fica mais fácil de *debugar* e testar o mesmo.

> **Nota**: Se você estiver utilizando *Windows* e um servidor *Firebird*, poderá acontecer um erro ao tentar conectar ao banco de dados: *Failed to locate host machine. Undefined service gds_db/tcp.*
>
> Você terá que abrir o arquivo *C:\WINDOWS\system32\drivers\etc\services* em um editor de texto e adicionar o registro do servidor *Firebird* na porta 3050, conforme o texto abaixo:
>
> gds_db 3050/tcp # Firebird

No código da **Listagem 3** atribui-se os resultados das funções *connect* a uma variável para verificar se a conexão teve sucesso ou não, o que é realizado através dos *if's*.

Listagem 2. Script para criação das tabelas no Firebird

```
CREATE TABLE 'CLIENTES'
(
  'ID' INTEGER NOT NULL,
  'NOME' VARCHAR(45) NOT NULL,
  'DATANASC' DATE NOT NULL,
  'FONE' VARCHAR(10),
  'EMAIL' VARCHAR(100),
 PRIMARY KEY ('ID')
);

CREATE GENERATOR 'GEN_ID_CLIENTE';
SET TERM ^ ;

CREATE TRIGGER 'TG_CLIENTES' FOR 'CLIENTES'
ACTIVE BEFORE INSERT POSITION 0
AS
begin
    NEW.ID= GEN_ID(gen_id_cliente, 1);
end
 ^
```

Listagem 3. Script PHP para testes de conexão aos bancos de dados

```
<?php
    $MYSQL = mysql_connect('localhost','root','root');
    $FIREBIRD = ibase_connect('localhost:C:\Direto.FDB','SYSDBA','masterkey');

    echo "<BR>";
    if ($MYSQL) {
      echo "MYSQL - CONECTADO";
    };
    echo "<BR>";
    if ($FIREBIRD) {
      echo "FIREBIRD - CONECTADO";
    };

    mysql_close($MYSQL);
    ibase_close($FIREBIRD);
?>
```

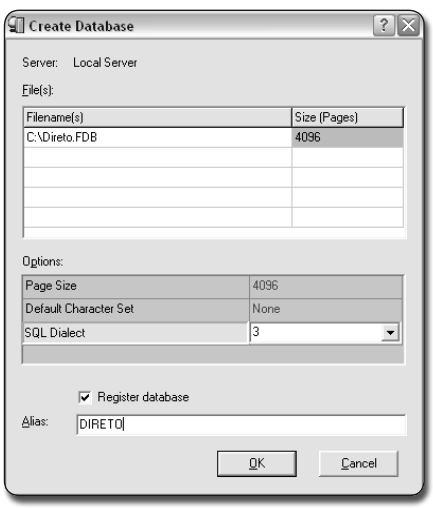


Figura 3. Criação do banco de dados Firebird no IBOConsole

Também utilizamos os comandos *mysql_close* e *ibase_close* para fecharmos as conexões, passando como parâmetro as variáveis criadas para atribuição do link da conexão, e, se houver falha essa variável terá como valor *FALSE*. O *close* é um comando opcional uma vez que o PHP libera todos os recursos no fechamento do Script.

É importante ressaltar a necessidade de fecharmos as conexões, não correndo o risco de esgotar o número de conexões simultâneas (parâmetro do *PHP* no arquivo *PHP.ini*), bem como recursos do servidor, quando, após um acesso ao banco de dados, você tem bastante código *PHP* para executar ainda. Principalmente se sua conexão não for persistente.

Listagem 4. Script com informações necessárias para conexões ao banco de dados

```
<?php
   $MServidor = "localhost";
   $MUsuario = "root";
   $MSenha = "root";

   $FServidor = "localhost:C:\Direto.FDB";
   $FUsuario = "SYSDBA";
   $FSenha = "masterkey";
?>
```

Listagem 5. Script para teste de conexão

```
<?php
    require_once('conexao.php');
    $MYSQL = mysql_pconnect($MServidor,$MUsuario,$MSenha);
    $FIREBIRD = ibase_pconnect($FServidor,$FUsuario,$FSenha);

    echo "<BR>";
    if ($MYSQL) {
      echo "MYSQL - CONECTADO";
    };
    echo "<BR>";
    if ($FIREBIRD) {
      echo "FIREBIRD - CONECTADO";
    };
?>
```

Conexões persistentes ao banco de dados

Tanto *MySQL* como *Firebird* possuem um comando para criar conexões persistentes ao banco de dados. O uso de conexões persistentes pode ser interessante em caso de muitos acessos simultâneos ao banco de dados. O que acontece com esse tipo de conexão é o reaproveitamento, uma vez que, a cada nova conexão, o sistema primeiro verifica se já não há uma conexão ao mesmo banco, com o mesmo usuário e senha. Então se você tem vários processos sendo executados ao mesmo tempo, todos poderão compartilhar uma única conexão ao banco de dados.

Um porém, nesse tipo de conexão é a dependência da forma como o *PHP* é executado no servidor. Se for como um *CGI* essa persistência não funciona, pois, a cada requisição as conexões serão instanciadas e no final, destruídas. Agora se o *PHP* funcionar como um módulo (grande maioria) você poderá ter muita eficiência nas conexões usando a persistência, pois depois da primeira conexão a sua reutilização será muito mais rápida que criar uma nova.

Usando a persistência o comando *close* (*mysql_close* e *ibase_close*) não irá mais fechar a conexão, pois a idéia é manter a mesma sempre ativa. Aí abre-se um precedente para termos alguns problemas quando precisamos travar algumas tabelas para edição, pois para destravar é necessário fechar a conexão.

O que muda para realizarmos uma conexão persistente? Somente a adição da letra "**p**" ao comando *connect*.

Mysql:

```
mysql_pconnect(SERVIDOR, USUARIO, SENHA);
```

Exemplo: mysql_pconnect('localhost', 'root', 'root');

Firebird:

```
ibase_pconnect(SERVIDOR E PATH, USUARIO, SENHA);
```

Exemplo: ibase_pconnect('localhost:C:\Direto.FDB','SYSDBA','masterkey');

Tente agora modificar o código da **Listagem 3** tornando suas conexões persistentes.

Arquivo com a definição de banco de dados

Agora pense você em uma aplicação que tenha inúmeros scripts *PHP* que fazem conexão ao banco e precisam das informações de servidor, usuário e senha. Você não poderá escrever essas informações em todos os arquivos. Precisa-se criar um arquivo para centralizar essas informações em um único local, facilitando assim a troca em qualquer momento.

Por isso é normal criar um arquivo *PHP* que contenha essas informações. Para exemplificar tem-se o código da **Listagem 4** onde estão as informações para nossas conexões e no código da **Listagem 5** tem-se o teste de conexão, buscando as informações do outro arquivo.

No código da **Listagem 4** simplesmente declarou-se variáveis (iniciam com **$**), definindo como seus valores, as informações necessárias para conectar no banco de dados. No código da **Listagem 5** utilizamos essas variáveis para executar o *connect*. O comando *require_once* serve para incluir o arquivo especificado entre parêntese no *script*. Em outras palavras o *require_once* junta os arquivos na hora de execução criando um único. Também, com a mesma idéia existe o comando *require* que difere pela não observância de o arquivo já ter sido incluído podendo ocorrer problemas, como redefinição de funções, valores de variáveis, etc. Prefira sempre utilizar o *require_once*.

Nota do DevMan

Tente utilizar suas conexões de forma persistente. Só opte por conexões não persistentes quando for necessário travar uma tabela ou ter blocos de transações que dependem um do outro. Utilizando conexões persistentes suas aplicações terão uma performance melhor.

Listagem 6. Script para localização de dados

```
<?php
    require_once('conexao.php');
    $MYSQL = mysql_pconnect($MServidor,$MUsuario,$MSenha);
    $FIREBIRD = ibase_pconnect($FServidor,$FUsuario,$FSenha);

    mysql_select_db('direto',$MYSQL);
    $SQL = "insert into clientes (id, nome, datanasc, fone, email)
      values (0,'Fabrício Desbessel', '1997-03-17', '1135122020','fabricio@fabricio.pro.br')";
    $MResultado = mysql_query($SQL, $MYSQL);
    $FResultado = ibase_query($SQL, $FIREBIRD);

    echo "<BR>";
    if ($MResultado) {
      echo "MYSQL - Dado inserido com sucesso";
    };
    echo "<BR>";
    if ($FResultado) {
      echo "FIREBIRD - Dado inserido com sucesso";
    };
?>
```

Listagem 7. Script para alteração de dados

```
<?php
    require_once('conexao.php');
    $MYSQL = mysql_pconnect($MServidor,$MUsuario,$MSenha);
    $FIREBIRD = ibase_pconnect($FServidor,$FUsuario,$FSenha);

    mysql_select_db('direto',$MYSQL);
    $SQL = "update clientes set nome='Fabrício Clube Delphi' where id=1";
    $MResultado = mysql_query($SQL, $MYSQL) or die(mysql_error());
    $FResultado = ibase_query($SQL, $FIREBIRD) or die(ibase_errmsg());

    echo "<BR>";
    if ($MResultado) {
      echo "MYSQL - Dado alterado com sucesso";
    };
    echo "<BR>";
    if ($FResultado) {
      echo "FIREBIRD - Dado alterado com sucesso";
    };
?>
```

Listagem 8. Script para exclusão de dados

```
<?php
    require_once('conexao.php');
    $MYSQL = mysql_pconnect($MServidor,$MUsuario,$MSenha);
    $FIREBIRD = ibase_pconnect($FServidor,$FUsuario,$FSenha);

    mysql_select_db('direto',$MYSQL);
    $SQL = "delete from clientes where id=1";
    $MResultado = mysql_query($SQL, $MYSQL) or die(mysql_error());
    $FResultado = ibase_query($SQL, $FIREBIRD) or die(ibase_errmsg());

    echo "<BR>";
    if ($MResultado) {
      echo "MYSQL - Dado excluído com sucesso";
    };
    echo "<BR>";
    if ($FResultado) {
      echo "FIREBIRD - Dado excluído com sucesso";
    };
?>
```

Para incluir um script PHP dentro de outro podemos também utilizar os comandos *include* e *include_once*. Eles funcionam da mesma forma que o *require* e *require_once*, porém se acontecer um erro, como, por exemplo, o arquivo não ser encontrado, ou até mesmo erro no arquivo a ser incluído, o *include* irá somente gerar uma mensagem de *Warning* (cuidado) executando o restante do *script*. Já o *require*, em caso de erro produzirá um *Fatal error* (erro fatal) impedindo a execução do script restante.

Portanto, em caso de conexão a banco de dados é muito interessante utilizar o *require* para incluir os valores necessários para conectar, pois, se acontecer um erro, devemos abordar a execução do *script*.

Manipulação de Dados

Para inserir, alterar ou excluir dados utiliza-se instruções SQL. Para executá-las utilizamos um comando *query* diferenciando o início do comando conforme o banco de dados utilizado.

Mysql:

```
mysql_query(INSTRUÇÃO SQL, CONEXÃO);
```

Exemplo: mysql_query("insert into clientes (id, nome, datanasc, fone, email) values (0,'Fabrício Desbessel', '1997-03-17', '1135122020','fabricio@fabricio.pro.br')", $MYSQL);

Firebird:

```
ibase_query(INSTRUÇÃO SQL, CONEXÃO);
```

Exemplo: ibase_query("insert into clientes (id, nome, datanasc, fone, email) values (0,'Fabrício Desbessel', '1997-03-17', '1135122020','fabricio@fabricio.pro.br')", $FIREBIRD);

Os comandos *mysql_query* e *ibase_query* possuem a mesma funcionalidade que é executar uma instrução SQL no banco de dados retornando um valor booleano conforme o sucesso da operação. Para testar digite e execute o código da **Listagem 6.**

O código da **Listagem 5** inicia com a inclusão do arquivo *conexao.php* que contém as informações necessárias para a conexão ao banco de dados. Em seguida são realizadas as conexões persistentes ao banco de dados *MySQL* e *Firebird*.

O comando *mysql_select_db* serve para selecionar o banco de dados, que no exemplo tem o nome "*direto*", utilizando, como segundo parâmetro a variável que contém o link da conexão ao banco. Esse comando não é necessário no *Firebird* porque na conexão já informa-se o nome do banco, através do nome do arquivo *.FDB*. Se você não utilizar esse comando o *MySQL* vai executar a SQL no banco de dados utilizado na última vez.

Para executar a *query* e saber se a mesma foi executada com sucesso atribuímos o seu retorno a uma variável que irá receber *TRUE* ou *FALSE*. Pode-se ainda mandar parar a execução caso aconteça algum erro, com o uso do *die* que é um comando de *exit* onde é possível exibir a mensagem de erro. Veja como ficaria o comando utilizando o *die*:

```
$MResultado = mysql_query($SQL, $MYSQL) or
  die(mysql_error());
$FResultado = ibase_query($SQL, $FIREBIRD) or
  die(ibase_errmsg ());
```

Utilizando *or die* o comando terá que ser executado com sucesso ou será executado um *exit*, abandonando a execução dos comandos restantes no script e ainda exibindo a mensagem de erro que é apresentada pelas funções *mysql_error* e *ibase_errmsg*.

Como todos SGBD's utilizam o padrão ANSI para SQL é possível escrever da mesma forma a instrução SQL o que resultou em uma única instrução no código da **Listagem 5** que foi armazenado em uma variável ($SQL) para ser reaproveitada.

Da mesma forma pode-se executar instruções SQL de *Update* e *Delete*, como é demonstrado nos códigos das **Listagens 7** e **8**, respectivamente.

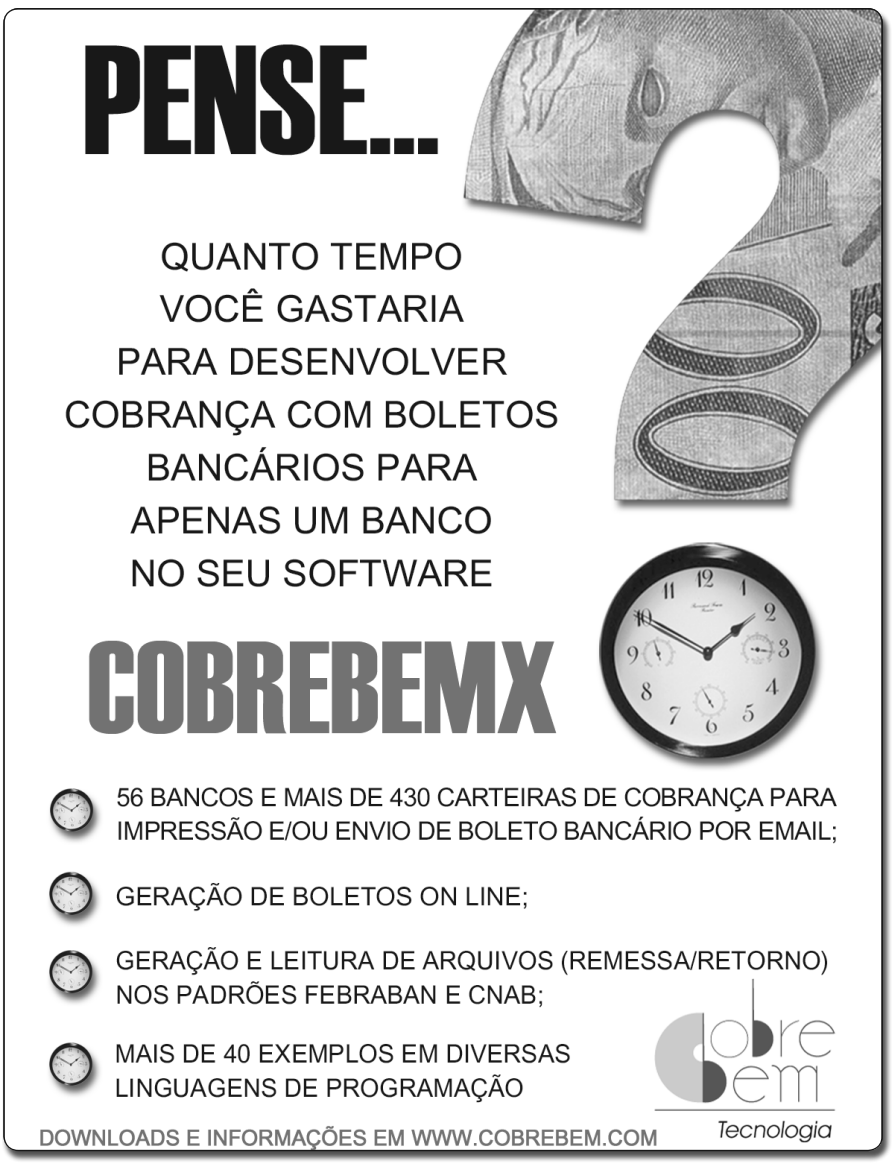
Para qualquer manipulação de dados pode-se utilizar o *query*, o que pode-se considerar muito positivo, pois facilita a criação de funções genéricas para abstrair as instruções SQL e até mesmo as conexões a banco de dados diferentes.

Seleção de Dados - Consultas

Para executar instruções SQL de *Select* utiliza-se também o comando *query*. Porém, faz-se necessário buscar linha por linha resultante do *select* associando os resultados em uma matriz, através do

Listagem 9. Script para consulta no banco de dados

```
<?php
    require_once('conexao.php');
    $MYSQL = mysql_pconnect($MServidor,$MUsuario,$MSenha);
    $FIREBIRD = ibase_pconnect($FServidor,$FUsuario,$FSenha);
    mysql_select_db('direto',$MYSQL);
    $SQL = "select * from clientes order by nome";
    $MResultado = mysql_query($SQL, $MYSQL);
    $FResultado = ibase_query($SQL, $FIREBIRD);
    echo "<BR>";
    echo "MYSQL - Dados selecionados com sucesso:<BR>";
    while ($MColuna = mysql_fetch_assoc($MResultado)) {
      echo $MColuna["id"]." - ";
      echo $MColuna["nome"]." - ";
      echo $MColuna["datanasc"]." - ";
      echo $MColuna["fone"]." - ";
      echo $MColuna["email"]."<BR>";
    }
    mysql_free_result($MResultado);
    echo "<BR>";
    echo "Firebird - Dados selecionados com sucesso:<BR>";
    while ($FColuna = ibase_fetch_assoc($FResultado)) {
      echo $FColuna["ID"]." - ";
      echo $FColuna["NOME"]." - ";
      echo $FColuna["DATANASC"]." - ";
      echo $FColuna["FONE"]." - ";
      echo $FColuna["EMAIL"]."<BR>";
    }
    ibase_free_result($FResultado);
?>
```

Listagem 10. Script de consulta otimizada no Firebird

```
<?php
    require_once('conexao.php');
    $FIREBIRD = ibase_pconnect($FServidor,$FUsuario,$FSenha);

    $SQL = "select * from clientes order by nome";
    $Consulta = ibase_prepare($FIREBIRD, $SQL);
    $FResultado = ibase_execute($Consulta);

    echo "<BR>";
    echo "Firebird - Dados selecionados com sucesso:<BR>";
    while ($FColuna = ibase_fetch_assoc($FResultado)) {
      echo $FColuna["id"]." - ";
      echo $FColuna["NOME"]." - ";
      echo $FColuna["DATANASC"]." - ";
      echo $FColuna["FONE"]." - ";
      echo $FColuna["EMAIL"]."<BR>";
    }
    ibase_free_result($FResultado);
?>
```

comando *fetch_assoc*. Veja na **Listagem 9** como fazer uma consulta e exibir as informações resultantes.

No código da **Listagem 8,** executa-se uma consulta em cada banco de dados, e usa-se um laço de repetição *while* para buscar os valores de cada linha resultante da consulta por meio da função *fetch_assoc* que retorna uma matriz com as informações da linha. Para acessar o valor de uma coluna, a forma mais simples é utilizar a variável passando como índice o nome da coluna. Como é possível em uma consulta ter nomes iguais para as colunas podemos usar a função *fetch_row* que faz o mesmo da função *fetch_assoc* só que como índice utiliza valores numéricos, na mesma ordem dos campos selecionados através da cláusula *select*. Assim pode-se utilizar, por exemplo *$FColuna[1]* para buscar a informação do campo 1.

Existe uma diferença significativa entre *MySQL* e *Firebird* na hora de exibir as informações. No *Firebird* foi necessário colocar o nome dos campos em maiúsculo. Isso porque o script de criação das tabelas (**Listagem 2**) estava todo maiúsculo. Para o PHP isso é muito significativo e sugere-se cuidar bastante a nomeação, tanto de campos como de nome de tabelas no banco de dados.

Ao final da execução da consulta faz-se necessário liberar a memória utilizada para guardar as informações resultantes, função do comando *free_result*. Acho que não é necessário, mas vou ressaltar: não pode-se esquecer de liberar a memória. Imagina você 100 mil registros selecionados em uma consulta e o desenvolvedor esquecer de liberar. Assim não tem servidor com recursos suficientes! Mesmo que depois da execução do *script* tudo seja liberado da memória isso é essencial, pois mesmo em frações de segundos, você pode esgotar os recursos do servidor, impedindo a execução com sucesso de outro script e, até mesmo, ter problemas no seu.

Otimizando consultas no Firebird

Quem conhece e utiliza o *Firebird* sabe que o mesmo possui formas de preparar as SQL e depois executá-las. Há um ganho muito grande de performance na preparação de uma consulta quando a mesma é utilizada repetidamente, o que acontece bastante em sites com muitos acessos. Essa otimização acontece porque o *Firebird* monta um plano de execução para a consulta e esse plano poderá ser reutilizado para uma consulta igual, diminuindo o tempo total de execução da consulta.

Na **Listagem 10** fazemos uma consulta no *Firebird* utilizando as funções *prepare* e *execute*, que proporcionam mais velocidade e otimização de recursos.

Olhando o código da **Listagem 10,** verifica-se o funcionamento das funções *prepare* e *execute* que são específicas do *Firebird*. No *prepare*, passa-se como parâmetro a conexão com o banco e a instrução SQL. Com isso o *Firebird* já cria o plano de execução da consulta. O *execute* necessita, como parâmetro, somente a variável que recebeu o retorno do *prepare*. O restante do código é igual, o que torna a utilização do *prepare* e *execute* uma obrigação em *script* que conectam ao *Firebird*, pois necessita pequenas modificações no código (duas linhas) e agiliza muito a consulta e, conseqüentemente, a execução do *script*.

Conclusão

Realmente é muito fácil manipular dados no *PHP*. Com apenas alguns comandos faz-se todas as operações necessárias em uma aplicação.

Também fica muito fácil você mesmo criar suas funções como, conecta, insere, altera, consulta, etc. Ainda melhor, é, se essas funções abstraírem a conexão com o banco de dados. Assim sua aplicação poderia ser multi-banco, o que seria um grande diferencial.

Agora está na hora de testar seu aprendizado. Coloque a mão na massa e crie suas próprias funções. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Conhecimento faz diferença!

Mais de 60 mil downloads na primeira edição!

Faça já sua assinatura digital! | *www.devmedia.com.br/es*

Faça um *up grade* em sua carreira.

Em um mercado cada vez mais focado em qualidade ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lança para os desenvolvedores brasileiros sua primeira revista digital totalmente especializada em Engenharia de Software. Todos os meses você irá encontrar artigos sobre Metodologias Ageis; Metodologias tradicionais (*document driven*); ALM (*application lifecycle management*); SOA (aplicações orientadas a servicos); Analise de sistemas; modelagem; Métricas; orientação à objetos; UML; testes e muito mais. **Assine já!**

DevMedia GROUP

A INVASÃO DE PROPRIEDADE NÃO SE LIMITA A RESIDÊNCIAS.

Assegure-se que seu software está protegido pelo líder em segurança da informação, antes que seja tarde demais.

SafeNet tem à sua disposição a proteção de software mais avançada e segura. As Chaves de Hardware Sentinel oferecem tecnologia que incorpora criptografia de chave pública, criptografia AES e autenticação interna para garantir o mais alto nível de proteção anti-pirataria.

Peça seu Kit de Desenvolvimento hoje mesmo.

Entre em contato em +55 11 4208-7700 em SP-Brasil
ou por e-mail infopirataria@safenet-inc.com

25 YEARS
1983–2008

SafeNet